УЧЕБНИК ДЛЯ ВУЗОВ

В. И. Юров

ASSEMBLER

2-е издание

Допущено Министерством образования Российской Федерации
в качестве учебного пособия для студентов высших учебных заведений,
обучающихся по направлению подготовки дипломированных
специалистов «Информатика и вычислительная техника»



300.piter.com

Издательская программа
300 лучших учебников для высшей школы
в честь 300-летия Санкт-Петербурга
осуществляется при поддержке Министерства образования РФ



Москва • Санкт-Петербург • Нижний Новгород • Воронеж
Ростов-на-Дону • Екатеринбург • Самара
Киев • Харьков • Минск
2003

ББК 32.973-018.1я7
УДК 681.3.06(075)
Ю70

Рецензенты:

Гурко А. В., кандидат технических наук, доцент кафедры военной кибернетики филиала ВУ ПВО

Тузов В. **А.**, доктор физико-математических наук, профессор кафедры «Технологии программирования» Санкт-Петербургского государственного университета

Ю70 Assembler. Учебник для вузов. 2-е изд. / В. И. **Юров** — СПб.: Питер, 2003. — 637 с.: ил.

ISBN **5-94723-581-1**

В учебнике рассматриваются вопросы программирования на языке ассемблера для компьютеров на базе микропроцессоров фирмы Intel. Основу **книги** составляет материал, **являющийся** частью курса, читаемого автором в высшем учебном заведении и посвященного вопросам системного программирования. По сравнению с первым изданием учебник существенно переработан. Исправлены ошибки и неточности. Добавлено описание команд для Intel-совместимых процессоров (до Pentium IV включительно).

Книга будет полезна студентам вузов, программистам и всем желающим изучить язык Assembler.

Допущено Министерством образования Российской Федерации в качестве учебного пособия для студентов высших учебных **заведений,** обучающихся по направлению подготовки дипломированных специалистов «Информатика и вычислительная техника».

ББК 32.973-018.1я7
УДК 681.3.06(075)

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 5-94723-581 -1 © ЗАО Издательский дом «Питер», 2003

Краткое содержание

Предисловие 13
Благодарности 16
От издательства 16
Глава 1. Организация современного компьютера 17
Глава 2. Программно-аппаратная архитектура IA-32 процессоров Intel 26
Глава 3. Система команд процессора IA-32 58
Глава 4. Системы счисления 74
Глава 5. Синтаксис ассемблера 85
Глава 6. Первая программа 121
Глава 7. Команды обмена данными 146
Глава 8. Арифметические команды 165
Глава 9. Логические команды и команды сдвига 193
Глава 10. Команды передачи управления 209
Глава 11. Программирование типовых управляющих структур 236
Глава 12. Цепочечные команды 249
Глава 13. Сложные структуры данных 268
Глава 14. Макросредства языка ассемблера 293
Глава 15. Модульное программирование 324

Глава 16. Создание Windows-приложений на ассемблере 365
Глава 17. Архитектура и программирование сопроцессора 447
Вместо заключения 510
Приложение. Система команд процессоров IA-32 511
Список литературы 625
Алфавитный указатель 626

Содержание

Предисловие ... 13
Благодарности ... 16
От издательства ... 16

Глава 1. Организация современного компьютера ... 17

Машинный язык и язык ассемблера ... 18
История процессоров Intel ... 21
Итоги ... 24

Глава 2. Программно-аппаратная архитектура IA-32 процессоров Intel ... 26

Архитектура ЭВМ ... 27
Архитектура IA-32 ... 29
Варианты микроархитектуры процессоров Intel ... 30
Программная модель IA-32 ... 40
Итоги ... 56

Глава 3. Система команд процессора IA-32 ... 58

Формат машинных команд IA-32 ... 59
Поле префиксов ... 61
Код операции ... 63
Байт режима адресации mod r/m ... 64
Байт масштаба, индекса и базы ... 70
Поля смещения и непосредственного операнда ... 71
Функциональная классификация машинных команд ... 72
Итоги ... 73

Глава 4. Системы счисления ... 74

Двоичная система счисления ... 75
Шестнадцатеричная система счисления ... 76
Десятичная система счисления ... 77
Перевод чисел из одной системы счисления в другую ... 78
Перевод в десятичную систему счисления ... 78
Перевод в двоичную систему счисления ... 78
Перевод в шестнадцатеричную систему счисления ... 79

Перевод дробных чисел ... 80
Перевод чисел со знаком ... 83
Итоги ... 84

Глава 5. Синтаксис ассемблера ... 85

Синтаксис ассемблера ... 86
Операнды ... 90
Операнды-выражения ... 97
Директивы сегментации ... 103
Простые типы данных ассемблера ... 110
Итоги ... 119

Главаб. Первая программа ... 121

Жизненный цикл программы ... 121
Пример простой программы ... 123
Процесс разработки программы ... 128
Трансляция программы ... 129
Компоновка программы ... 134
Отладка программы ... 135
Особенности разработки программ в MASM ... 141
Особенности разработки программ в Microsoft Visual Studio ... 142
Выбор пакета ассемблера ... 143
Итоги ... 144

Глава 7. Команды обмена данными ... 146

Пересылка данных ... 147
Ввод из порта и вывод в порт ... 149
Работа с адресами и указателями ... 156
Преобразование данных ... 158
Работа со стеком ... 160
Итоги ... 164

Глава 8. Арифметические команды ... 165

Обзор ... 166
Целые двоичные числа ... 167
Десятичные числа ... 168
Арифметические операции над целыми двоичными числами ... 170
Сложение двоичных чисел без знака ... 170
Сложение двоичных чисел со знаком ... 172
Вычитание двоичных чисел без знака ... 173
Вычитание двоичных чисел со знаком ... 175
Вычитание и сложение операндов большой размерности ... 176
Умножение двоичных чисел без знака ... 177
Умножение двоичных чисел со знаком ... 179
Деление двоичных чисел без знака ... 179
Деление двоичных чисел со знаком ... 180
Вспомогательные команды для арифметических вычислений ... 181
Команды преобразования типов ... 181
Другие полезные команды ... 183
Арифметические операции над двоично-десятичными числами ... 183

Неупакованные BCD-числа ... 184
Упакованные BCD-числа ... 190
Итоги ... 192

Глава 9. Логические команды и команды сдвига ... 193

Логические данные ... 194
Логические команды ... 195
Команды сдвига ... 199
Линейный сдвиг ... 199
Циклический сдвиг ... 201
Дополнительные команды сдвига ... 203
Примеры работы с битовыми строками ... 205
Рассогласование битовых строк ... 205
Вставка битовых строк ... 206
Извлечение битовых строк ... 207
Пересылка битов ... 207
Итоги ... 207

Глава 10. Команды передачи управления ... 209

Безусловные переходы ... 214
Команда безусловного перехода ... 215
Процедуры ... 219
Условные переходы ... 224
Команда сравнения ... 225
Команды условного перехода и флаги ... 227
Команды условного перехода и регистр ECX/CX ... 228
Установка байта по условию ... 229
Организация циклов ... 229
Ограничение области видимости для меток ... 234
Итоги ... 235

Глава 11. Программирование типовых управляющих структур ... 236

Условный оператор if-else ... 237
Оператор выбора switch ... 237
Операторы цикла ... 240
Оператор цикла с предусловием while ... 240
Операторы continue и break ... 240
Оператор цикла с постусловием do-while ... 241
Оператор итерационного цикла for ... 241
Особенности пакета MASM ... 242
Условная конструкция .IF ... 242
Циклическая конструкция .REPEAT ... 243
Циклическая конструкция .WHILE ... 244
Конструкции .BREAK и .CONTINUE ... 244
Комплексный пример ... 245
Итоги ... 248

Глава 12. Цепочечные команды ... 249

Пересылка цепочек ... 253
Команда MOVS ... 253
Пересылка байтов, слов и двойных слов ... 254

Сравнение цепочек ... 255
Команда CMPS ... 255
Сравнение байтов, слов и двойных слов ... 258
Сканирование цепочек ... 259
Команда SCAS ... 259
Сканирование строки байтов, слов, двойных слов ... 260
Загрузка элемента цепочки в аккумулятор ... 261
Команда LODS ... 261
Загрузка в регистр AL/AX/EAX байтов, слов, двойных слов ... 262
Перенос элемента из аккумулятора в цепочку ... 262
Команда STOS ... 263
Сохранение в цепочке байта, слова, двойного слова из регистра AL/AX/EAX ... 264
Работас портами ввода-вывода ... 265
Ввод элемента цепочки из порта ввода-вывода ... 265
Вывод элемента цепочки в порт ввода-вывода ... 265
Итоги ... 266

Глава 13. Сложные структуры данных ... 268

Массивы ... 269
Описание и инициализация массива в программе ... 269
Доступ кэлементам массива ... 271
Двухмерные массивы ... 275
Типовые операции с массивами ... 278
Структуры ... 280
Описание шаблона структуры ... 281
Определение данных с типом структуры ... 281
Методы работы со структурами ... 282
Объединения ... 284
Записи ... 286
Описание записи ... 286
Определение экземпляра записи ... 287
Работа с записями ... 288
Дополнительные возможности обработки ... 290
Итоги ... 292

Глава 14. Макросредства языка ассемблера ... 293

Псевдооператоры EQU и = ... 295
Макрокоманды ... 297
Макродирективы ... 305
Директивы WHILE и REPT ... 305
Директива IRP ... 307
Директива IRPC ... 307
Директивы условной компиляции ... 308
Директивы компиляции по условию ... 308
Директивы генерации ошибок ... 315
Константные выражения в условных директивах ... 318
Дополнительные средства управления трансляцией ... 319
Директивы управления файлом листинга ... 321
Итоги ... 322

Глава 15. Модульное программирование ... 324

Структурное программирование ... 325
Процедуры в языке ассемблера ... 327
Передача аргументов через регистры ... 331
Передача аргументов через общую область памяти ... 332
Передача аргументов через стек ... 334
Использование директив EXTRN и PUBLIC ... 337
Возврат результата из процедуры ... 340
Связь ассемблера с языками высокого уровня ... 344
Встраиваемый ассемблерный код ... 345
Внешний ассемблерный код ... 346
Команды ENTER и LEAVE ... 356
С и ассемблер ... 359
Итоги ... 364

Глава 16. Создание Windows-приложений на ассемблере ... 365

Программирование оконных Windows-приложений ... 367
Каркасное Windows-приложение на C/C++ ... 368
Каркасное Windows-приложение на ассемблере ... 379
Стартовый код ... 387
Главная функция ... 389
Обработка сообщений в оконной функции ... 398
Средства TASM для разработки Windows-приложений ... 401
Углубленное программирование на ассемблере для Win32 ... 404
Ресурсы Windows-приложений на языке ассемблера ... 406
Меню в Windows-приложениях ... 406
Перерисовка изображения ... 413
Окна диалога в Windows-приложениях ... 420
Программирование консольных Windows-приложений ... 435
Минимальная программа консольного приложения ... 437
Организация высокоуровневого консольного ввода-вывода ... 438
Пример программы консольного ввода-вывода ... 440
Итоги ... 446

Глава 17. Архитектура и программирование сопроцессора ... 447

Архитектура сопроцессора ... 448
Регистр состояния SWR ... 453
Регистр управления CWR ... 454
Регистр тегов TWR ... 455
Форматы данных ... 456
Двоичные целые числа ... 457
Упакованные целые десятичные числа ... 457
Вещественные числа ... 458
Специальные численные значения ... 463
Система команд сопроцессора ... 466
Команды передачи данных ... 468
Команды загрузки констант ... 471
Команды сравнения данных ... 472

Арифметические команды ... 474
Команды трансцендентных функций ... 482
Команды управления сопроцессором ... 494
Исключения сопроцессора и их обработка ... 500
Недействительная операция ... 501
Деление на ноль ... 501
Денормализация операнда ... 501
Переполнение и антипереполнение ... 502
Неточный результат ... 502
Немаскируемая обработка исключений ... 502
Использование отладчика ... 505
Общие рекомендации по программированию сопроцессора ... 507
Итоги ... 508

Вместо заключения ... 510

Приложение. Система команд процессоров IA-32 ... 511

Целочисленные команды ... 513
Команды сопроцессора ... 553
Команды блока MMX ... 569
Команды блока XMM (SSE и SSE2) ... 587

Список литературы ... 625

Алфавитный указатель ... 626

Предисловие

Перед вами второе, исправленное и дополненное, издание книги «Assembler» из серии книг «Учебное пособие» издательства «Питер». Материал подобран и выстроен таким образом, чтобы служить информационно-методической базой для самостоятельного и организованного в рамках различных вузовских дисциплин и факультативов изучения языка ассемблера. Исходя из этого, учебник адресован следующим категориям читателей:

- преподавателям и студентам вузов для использования в качестве источника методически подобранной и систематизированной информации по различным аспектам применения ассемблера в контексте архитектуры Intel-совместимых процессоров;
- специалистам, занимающимся программированием и желающим освоить ассемблер для расширения своего профессионального кругозора, придания новых свойств своему опыту и для решения конкретных практических задач;
- школьникам, углубленно изучающим программирование для компьютеров на базе Intel-совместимых процессоров, а также всем тем, кто интересуется различными аспектами низкоуровневого программирования.

Материал учебника, выбранный уровень и методика его изложения преследуют следующие цели:

- изложить основы архитектуры Intel-совместимых процессоров;
- показать неразрывную связь архитектуры процессора с его машинным языком;
- представить систему машинных команд в виде функциональных групп с тем, чтобы объяснить цели, которые преследовали разработчики процессора при введении той или иной команды в систему машинных команд;
- научить использовать инструментальные средства разработки ассемблерных программ;
- научить осмысленному подходу к выбору средств ассемблера для реализации практических задач средней сложности.

Язык ассемблера является символическим представлением машинного языка, он неразрывно связан с архитектурой самого процессора. По мере внесения изменений в архитектуру процессора совершенствуется и сам язык ассемблера. По этой причине книга направлена на решение комплексной задачи — не просто рассмотреть ассемблер как еще один из десятков языков программирования, а показать

объективность его существования и неразрывную связь его конструкций с архитектурой конкретного процессора. Материал книги содержит описание основных особенностей архитектуры и системы команд процессоров Pentium Pro/MMX/II/III/IV.

Изложение материала в учебнике ведется в форме глав, которых всего 17. Логически их можно разделить на четыре части.

- В первых шести главах приводятся сведения о том, что представляет собой современный компьютер, что подразумевают понятия архитектуры процессора и компьютера в целом. Приводится информация о системе и синтаксисе машинных команд и команд ассемблера. Важными являются сведения о жизненном цикле типовой программы на ассемблере и инструментальных средствах разработки ассемблерных программ.
- Вторая часть книги, начиная с главы 7, посвящена рассмотрению команд ассемблера в соответствии с их функциональным назначением. Этот материал является базовым, и его достаточно, чтобы научиться писать простые программы на ассемблере. Начиная с главы 11 обсуждаются специальные средства ассемблера, которые используются для написания программ средней сложности. Рассматриваются возможности ассемблера для работы со сложными структурами данных, механизм макрокоманд, вопросы организации модульного программирования, в том числе принципы связи с модулями, написанными на C/C++ и Pascal.
- Последние две главы учебника посвящены различным аспектам написания ассемблерных программ, использующих современные программно-аппаратные расширения. Приводятся подробные сведения о порядке разработки оконных и консольных Windows-приложений, применении сопроцессора.
- Важная часть учебника — его приложение. В нем собрана справочная информация о командах (вплоть до Pentium IV). Данных, которые приведены в приложении, достаточно для проведения широкого круга работ — от общего знакомства с системой машинных команд до поддержки процесса дизассемблирования на уровне машинных кодов.
- В других приложениях, расположенных на сайте http://www.piter.com/download, собрана справочная информация о различных средствах пакетов ассемблера MASM и TASM.

Таким образом, книга является самодостаточным учебным пособием. Ее использование позволяет сформировать фундаментальные знания по различным аспектам низкоуровневого программирования на языке ассемблера для Intel-совместимых компьютеров. В долгосрочной перспективе материал учебника может служить справочником.

По сравнению с первым изданием в учебнике сделаны многочисленные изменения. Во-первых, материал был приведен в соответствие с современным уровнем развития процессоров Intel (до Pentium IV). Во-вторых, исправлены ошибки и опечатки. Во многом это заслуга внимательных читателей, за что им отдельная благодарность. В-третьих, произведено перестроение материала с учетом существования своеобразных продолжений учебников — книг серии «Практика» издательства

«Питер». Для настоящего учебника пока существует одна такая книга — «Assembler: практика». Ее содержание посвящено углубленному изучению языка ассемблера на различных востребованных на практике задачах прикладного характера. Материал книги «Assembler: практика» может служить основой для выполнения курсовых и дипломных работ, не говоря уже о его использовании при работе над различными программными проектами. Планируется появление второй книги из этой серии, которая будет ориентирована на освещение вопросов системного программирования. Каждую из этих книг следует позиционировать следующим образом. Учебник формирует основы и принципы, а на определенном этапе начинает выполнять справочные функции. Книги серии «Практикум» представляют описания вариантов реализации востребованных на практике задач прикладного и системного характера.

Об ошибках следует сказать отдельные слова. Конечно, книги, а тем более учебники, не должны содержать неточностей и ошибок, но при нынешних темпах жизни и развития техники это — недостижимый идеал. Конечно, это не говорит о том, что нельзя написать книгу без ошибок. Безусловно можно, но для этого потребуется времени в несколько раз больше и ее выход в свет может оказаться совершенно бессмысленной затратой времени и сил, так как к этому моменту перестанет быть актуальным сам предмет, которому посвящена книга. Поэтому вопрос о том, что важнее — своевременная книга, содержащая определенное количество ошибок, но помогающая читателю решить актуальные проблемы сегодняшнего дня, или идеально выверенное издание, освещающее вопросы вчерашнего дня, остается философским. Книги, в которых много исходного кода, — это особый вид книг, которые можно охарактеризовать как «книги-программы», а в программах, как известно, последних ошибок не бывает. Более того, рискну высказать мнение, что с учебной целью ошибки даже полезны. Это подтверждает и свой, и чужой опыт. Пословица «На ошибках учатся» имеет скрытый смысл — наиболее устойчивые знания формируются именно при исправлении своих и чужих ошибок. Это даже не знания, это уже профессиональный опыт. Тем не менее мои рассуждения не следует рассматривать как оправдание ошибок первого издания учебника и будущих ошибок второго издания. Они неизбежны, и автор будет благодарен всем читателям, кто заметит ошибки, неточности и просто опечатки и сообщит о них редакции или автору по указанным далее адресам электронной почты.

Часто спрашивают, для программирования каких процессоров можно использовать учебник. Ответ — для Intel-совместимых процессоров. Под термином «Intel-совместимые процессоры» подразумеваются процессоры фирм Intel, AMD, VIA, Transmeta, полностью поддерживающие базовую систему команд процессоров Intel и полностью или частично поддерживающие различные расширения базовой системы команд процессоров Intel.

Что нужно для работы с книгой? Во-первых, компьютер на базе Intel-совместимого процессора. Во-вторых, пакеты ассемблеров TASM и MASM. Причем лучше всего будет на этапе изучения использовать оба этих пакета. К сожалению, пакет TASM в самостоятельном виде уже не развивается, но работа с ним на этапе обучения достаточно комфортна, тем более что он имеет режим работы, позволяющий во многих случаях без дополнительной доработки переносить программы для

использования с пакетом MASM. Последняя доступная версия этого пакета — TASM 5.0. С пакетом MASM дела обстоят лучше — он развивается. Последние доступные версии — MASM 6.14 и MASM32 версии 7.0. И наконец, для работы нужен один из текстовых редакторов класса notepad.exe.

Благодарности

Хорошая традиция — выражение благодарности окружающим людям за их активный или пассивный вклад в появление книг на свет. Это не является некой формой похвальбы, а говорит лишь о том, что ты не один на белом свете и своими успехами и неудачам обязан многим людям, которых ты, возможно, никогда не видел и никогда не увидишь. Поэтому я рад и благодарен письму каждого читателя. Адрес электронной почты v_yurov@mail.ru всегда доступен для писем читателей относительно содержимого книг, пожеланий, сообщений о замеченных ошибках и неточностях. Особую благодарность выражаю жене Елене и детям — Саше и Юле.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Все исходные тексты программ, приведенные в книге, а также дополнительные приложения вы сможете найти по адресу http://www.piter.com/download.

Подробную информацию о наших книгах вы найдете по адресу http://www.piter.com.

Глава 1

Организация современного компьютера

- **Внешний вид типичного современного компьютера**
- **Структурная схема компьютера**
- **Место ассемблера**
- **Историческая ретроспектива процессоров Intel**

Современному человеку трудно представить свою жизнь без *электронно-вычислительных машин* (ЭВМ). В настоящее время любому желающему по силам собрать у себя на рабочем столе полноценный вычислительный центр, степень функциональности которого может быть ограничена только фантазией и финансовыми возможностями его обладателя.

В минимальной комплектации типовой персональный компьютер состоит из компонентов, показанных на рис. 1.1.

Из рисунка видно, что компьютер образуют несколько физических устройств, каждое из которых подключено к одному блоку, называемому *системным*. Если рассуждать логически, то ясно, что он играет роль некоторого координирующего устройства. Попытка открыть корпус и заглянуть внутрь системного блока, скорее всего, не приведет к удовлетворению жажды познания — непонятная совокупность разнообразных плат, блоков, индикаторов и соединительных проводов вряд ли даст однозначные ответы на возможные вопросы. Внутри монитора вы тем более ничего интересного не найдете, за исключением возможности удовлетворить другую жажду — жажду приключений. Если хорошо поискать отверткой подходящий источник, то, в отличие от системного блока, можно довольно быстро получить разряд в несколько киловольт.



Рис. 1.1. Компьютер и периферийные устройства

Рассмотрим структурную схему типичного современного персонального компьютера. Она не претендует на безусловную точность и имеет целью лишь показать назначение, взаимосвязь и типовой состав его элементов.

На рис. 1.2 показана функциональная схема системного блока компьютера на базе процессоров семейства Intel. На схеме представлены: центральный процессор, оперативная память, внешние устройства. Все компоненты соединены между собой через системную шину. Системная шина имеет дополнительную шину — шину расширения. В компьютерах на базе Pentium в качестве такой шины используется шина PCI (Peripheral Component Interface), к которой подсоединяются внешние устройства, а также шины более ранних стандартов, например ISA (Industry Standard Architecture).

На рисунке показана самая общая схема сердца компьютера — процессора. Основу процессора составляют блок микропрограммного управления, исполнительное устройство, обозначенное как «конвейер», и регистры. Остальные компоненты процессора выполняют вспомогательные функции. Более подробный вариант этой схемы мы рассмотрим в следующей главе.

Машинный язык и язык ассемблера

Чтобы лучше понять принципы работы компьютера, давайте сравним его с человеком. У компьютера есть органы восприятия информации из внешнего мира — это клавиатура, мышь, накопители на магнитных дисках (на схеме они расположены под системными шинами). У компьютера есть органы, «переваривающие» полученную информацию, — это центральный процессор и оперативная память. И наконец, у компьютера есть органы речи, выдающие результаты переработки. Это также некоторые из устройств ввода-вывода, расположенные в нижней части схемы. Современным компьютерам, конечно, далеко до человека. Их можно сравнить с существами, взаимодействующими с внешним миром на уровне большого, но ограниченного набора безусловных рефлексов. Этот набор рефлексов образует *систему машинных команд.* На каком бы высоком уровне вы ни общались с компьютером, в конечном итоге все сводится к скучной и однообразной последователь-

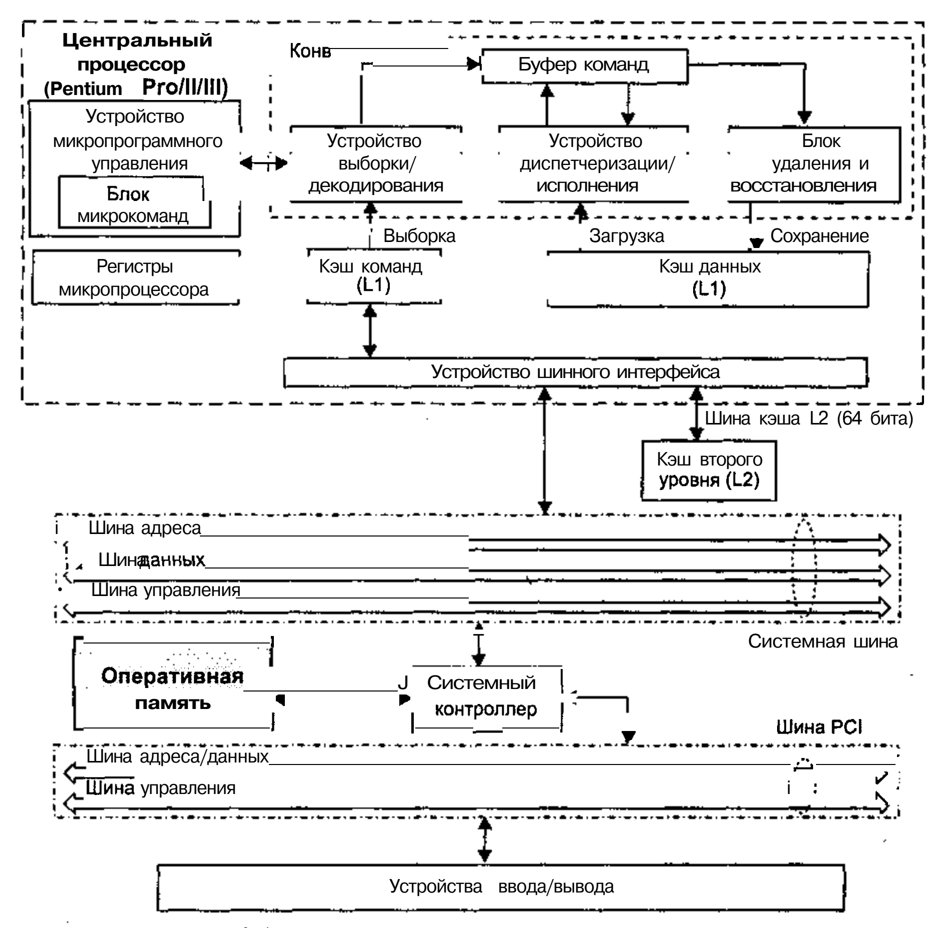


Рис. 1.2. Структурная схема персонального компьютера

ности машинных команд. Каждая машинная команда является своего рода раздражителем для возбуждения того или иного безусловного рефлекса. Реакция на этот раздражитель всегда однозначна и «зашита» в блоке микропрограммного управления в виде микропрограммы. Эта микропрограмма и реализует действия по выполнению машинной команды, но уже на уровне сигналов, подаваемых на те или иные логические схемы компьютера, тем самым управляя различными подсистемами компьютера. В этом состоит так называемый принцип микропрограммного управления. Продолжая аналогию с человеком, отметим: для того чтобы компьютер правильно «питался», придумано множество операционных систем, компиляторов сотен языков программирования и т. д. Но все они являются, по сути, лишь блюдом, на котором по определенным правилам доставляется пища (программы) желудку (компьютеру). Только (вот досада!) желудок компьютера любит диетическую, однообразную пищу — подавай ему информацию структурированную, в виде строго организованных последовательностей нулей и единиц, комбинации которых и составляют *машинный язык*.

Таким образом, внешне являясь полиглотом, компьютер понимает только один язык — язык машинных команд. Конечно, для общения и работы с компьютером необязательно знать этот язык, но практически любой профессиональный программист рано или поздно сталкивается с необходимостью его изучения. К счастью, при этом человеку не нужно пытаться постичь значение различных комбинаций двоичных чисел, так как еще в 50-е гг. программисты стали использовать для программирования символический аналог машинного языка, который назвали *языком ассемблера.* Этот язык точно отражает все особенности машинного языка. Именно поэтому, в отличие от языков высокого уровня, язык ассемблера для каждого типа компьютеров свой. Более того, бессмысленны разговоры о том, что ассемблер как язык программирования устарел и знать его необязательно. АССЕМБЛЕР ОБЪЕКТИВЕН, и подобные разговоры в определенной ситуации могут выглядеть довольно глупо (особенно для особо «продвинутых» программистов).

На первый взгляд ассемблер кажется очень сложным, но это не так. Для освоения этого языка совершенно не нужно выучивать наизусть все его команды и директивы. Более того, в большинстве случаев для практической работы достаточно понимания основных концепций и идей, лежащих в основе языка ассемблера. Детали реализации той или иной команды всегда можно найти в справочнике команд, гораздо важнее понимать, какое место данная команда занимает в системе команд, в идеале хорошо было бы знать и цели, которые преследовали разработчики процессора, вводя данную команду в систему машинных команд. Одна из целей данного учебника — сформировать у читателя именно такое понимание языка ассемблера.

Из всего сказанного напрашивается вывод о том, что самую эффективную программу можно написать только на ассемблере (при условии, что ее пишет квалифицированный программист), так как этот язык является «родным» для компьютера. Здесь есть одно маленькое «но»: это очень трудоемкий и требующий большого внимания и практического опыта процесс. Поэтому реально на ассемблере пишут в основном программы, которые должны обеспечить эффективную работу с аппаратной частью компьютера. Иногда на ассемблере пишутся критичные ко времени выполнения или расходованию памяти фрагменты программы. Впоследствии они оформляются в виде подпрограмм и совмещаются с кодом на языке высокого уровня.

В данном учебнике будут рассмотрены два фундаментальных для понимания логики функционирования компьютера вопроса: первый — архитектурные особенности процессора и основы его взаимоотношений с другими компонентами компьютера; второй — место и роль языка ассемблера во всем этом процессе. За основу в нашем рассмотрении будут взяты процессоры фирмы Intel. Необходимо отметить, что эти процессоры не являются единственными процессорами на рынке *аппаратного обеспечения* (hardware), хотя и занимают довольно большой сегмент этого рынка. Исторически сложилось так, что архитектура процессоров Intel полностью или частично поддерживается процессорами других фирм. Поэтому процессорам фирмы Intel приходится делить свой сегмент рынка с процессорами фирм AMD, VIA, Transmeta. Так как в своем сегменте рынка (Intel-совместимых процессоров) процессоры фирмы Intel являются стандартом де-факто, то в данном

учебнике речь будет идти исключительно о них, однако все приводимые примеры программ будут работать и на Intel-совместимых процессорах процессорах других фирм.

История процессоров Intel

Процессоры Intel в своем развитии прошли довольно сложный и насыщенный различными событиями путь. Корпорация Intel была основана в 1968 г. Первый процессор i4004 был разработан Intel в 1969 г. Он представлял собой 4-разрядное параллельное вычислительное устройство с 2300 транзисторами. Его возможности были сильно ограничены — он выполнял всего лишь четыре основные арифметические операции. Поначалу i4004 применялся в карманных калькуляторах. Позднее сфера его использования была расширена за счет различных систем управления, в частности, систем управления светофорами.

В 1972 г. был выпущен 8-разрядный процессор i8008 с адресацией внешней оперативной памяти размером 16 Кбайт. Неожиданный успех этого процессора подтолкнул разработчиков Intel к дальнейшим изысканиям. В 1974 г. был выпущен i8080, который, оставаясь 8-разрядным, мог адресовать 64 Кбайт оперативной памяти. Для того времени это было революцией, этот процессор был выпущен в миллионах экземпляров. Историю современных Intel-совместимых процессоров принято вести именно от i8080.

Очередной революционный процессор Intel — i8086 — появился в 1978 г. Его основные характеристики — наличие 16-разрядных регистров, 16-разрядной шины данных. Поддержка сегментной организации памяти наряду с 20-разрядной шиной адреса позволяла организовать адресацию памяти в пределах 1 Мбайт при доступности адресного пространства размером 256 Кбайт. С целью занятия более широкой ниши на рынке Intel вскоре представила более дешевый вариант этого процессора — i8088. При внутренней 16-разрядной архитектуре i8086 он имел 8-разрядную шину данных, вследствие чего был менее производителен. Важно отметить, что дешевизна достигалась в основном не за счет дешевизны самого процессора, а за счет удешевления конечного продукта — компьютера на его основе. Другой причиной появления и широкого распространения i8088 было наличие большого количества 8-разрядных внешних устройств. 8-разрядная шина данных i8088 позволяла упростить процесс сопряжения с этими устройствами. Отметим, что одним из внешних устройств, поддерживаемых процессорами i8086/88, был математический сопроцессор i8087, который мог устанавливаться в специальный разъем материнской платы. Это позволяло более эффективно выполнять операции с плавающей точкой в соответствии со стандартом IEEE-754. Важно также подчеркнуть, что, начиная с i8086/88, все процессоры Intel совместимы «снизу вверх». Следствием этого является гарантированная возможность запуска на современном процессоре Pentium практически любой программы, написанной для i8086/88 (естественно, при соответствующей поддержке системного программного обеспечения).

Согласно известному закону, *программное обеспечение* (software), подобно газу, стремится заполнить весь доступный ему объем. Поэтому поначалу кажущиеся

фантастическими возможности новых процессоров со временем становятся узами, тем самым подталкивая конструкторов процессоров искать новые архитектурные и технологические решения для перевода программ в новые, более свободные рамки существования. Оперативная память объемом 1 Мбайт — это много. Долгая жизнь процессоров i8086/88 подтверждает это. Но с течением времени даже такой объем памяти превратился в ограничение, в связи с чем стали применяться различные технологии для обхода этого ограничения. Многие, наверное, еще помнят загрузку MS-DOS с драйверами расширенной памяти EMS (Expanded Memory Specification), с помощью которых через окно размером 64 Кбайт можно было обращаться к внешней дополнительной памяти размером до 32 Мбайт. Но это не могло продолжаться вечно, и в 1982 г. Intel представляет процессор i80286. И хотя он не решил окончательно проблему ограничения пространства памяти, в нем были реализованы определенные архитектурные изменения, благодаря которым последующие модели процессоров позволили разрядить ситуацию с доступом к памяти. К главным нововведениям следует отнести расширение шины адреса до 24 битов, что позволило адресовать уже 16 Мбайт оперативной памяти, а также появление еще одного режима работы процессора — защищенного. В этом отношении данный процессор также можно считать революционным. До этого в процессорах Intel отсутствовала поддержка на процессорном уровне какой-либо защиты программ от взаимного влияния. Введение подобных средств в архитектуру процессоров Intel стимулировало производителей на разработку многозадачных операционных систем. Наиболее яркие примеры — Microsoft (ОС Windows) и IBM (OS/2). Справедливости ради следует отметить, что доминирующей системой для i286 была MS-DOS, и данный процессор использовался как более быстрый вариант i8086/88. Для широкого распространения многозадачные системы должны были пройти определенный путь развития.

Мощным стимулом к развитию многозадачных систем стали возможности, предоставляемые новым процессором i80386, выпущенным Intel в 1985 г. Это был первый 32-разрядный процессор, который положил начало семейству процессоров IA-32 (32-bit Intel Architecture). Главные отличительные особенности: 32-разрядные шины адреса и данных (адресация оперативной памяти — до 4 Гбайт); увеличенная до 32 битов разрядность внутренних регистров процессора; введение третьего режима работы процессора (режима виртуального процессора 8086); поддержка страничной адресации памяти, благодаря которой стало возможном за счет дисковой памяти виртуализировать доступ к оперативной памяти и теоретически расширить ее рамки вплоть до 4 Гбайт. Аналогично ситуации с i8086 и i8088, одновременно был выпущен более дешевый вариант процессора i80386 (i80386-DX) с 16-разрядной шиной данных — i80386-SX. При необходимости на материнскую плату можно было установить и математический сопроцессор i80387. Процессор i80386 первым из семейства процессоров Intel стал поддерживать возможность параллельного функционирования своих устройств: *устройства шинного интерфейса* (bus interface unit), *блока предварительной выборки* (code prefetch unit), *блока декодирования команд* (instruction decode unit), *исполнительного блока* (execution unit), *блока сегментации* (segment unit), *блока страничной адресации* (paging unit). Концепция параллельного функционирования устройств позже, а именно в i80486, стала основой другой концепции — конвейеризации вычислений.

Процессор i80486 появился в 1989 г. Основные его характеристики — наличие встроенного математического сопроцессора, поддержка многопроцессорного режима работы и работы с двумя видами кэш-памяти — внутренней размером 8 Кбайт (1-го уровня L1) и внешней кэш-памятью (2-го уровня L2). Важные изменения в архитектуре процессора i80486 коснулись концепции параллельных вычислений. Они стали воплощением идей, лежащих в основе RISC-технологии разработки процессоров. Согласно им, *машинные команды со сложным алгоритмом работы* (Complete-Instuction-Set Computing, CISC) на микропрограммном уровне реализовывались на основе более простых *команд из ограниченного набора* (Reduced Instruction Set Computer, RISC). Для выполнения CISC-команд требуется несколько тактов работы процессора, иногда несколько десятков. RISC-команды должны выполняться за один такт. Такое изменение внутреннего представления внешних команд, наряду с изменением последовательности их выполнения и режимов декодирования, позволило реализовать реальную конвейеризацию вычислений. В результате выполнение CISC-команды происходило за один такт процессора. Конвейер процессора i80486 состоял из 5 ступеней. Начиная с процессора i80486 процессоры Intel поддерживают различные концепции энергосбережения. Интересно, что совершенствование процессора i80486 шло в ходе его промышленного производства. Вследствие этого по своим возможностям следующие по времени выпуска процессоры i80486 в большей или меньшей степени отличались от предыдущих.

Ну и, наконец, ЭПОХА PENTIUM. Знаменитый своей ошибкой блока с плавающей точкой первый Pentium — Pentium-60 — был представлен в начале 1993 г. Благодаря суперскалярной архитектуре (с двумя конвейерами, известными как и и v) он мог выполнять две машинные инструкции за один такт. К внутреннему кэшу команд добавлен внутренний 8-килобайтный кэш данных. Реализована технология *предсказания переходов* (branch prediction). Для увеличения пропускной способности при обработке данных процессор Pentium поддерживает внутренние пути шириной 128 и 256 битов, внешняя шина данных увеличена до 64 битов. Добавлены средства для построения многопроцессорных систем, в частности *расширенный программируемый контроллер прерываний* (Advanced Programmable Interrupt Controller, APIC), дополнительные выводы и специальный режим *дуальной обработки* (dual processing), ориентированный на построение двухпроцессорных систем. Начиная с модели процессоров Pentium с тактовой частотой 133 МГц (1997 г.) в них был введен блок MMX-команд (MMX означает MultiMedia extensions). Реализованный на базе сопроцессора, данный блок поддерживал SIMD-технологию, которая предполагает обработку блока однородных данных одной машинной инструкцией. Цель введения данного расширения — увеличение производительности приложений, обрабатывающих массивы однородных целочисленных данных. Примеры таких приложений — программы обработки изображений, звука, архивирования-разархивирования данных и др. Излишне говорить, что все эти нововведения резко подняли производительность процессора Pentium по сравнению с его предшественником — процессором i486 — и не оставили последнему особых альтернатив для существования.

На сегодняшний день имеется три поколения, или семейства, процессоров Pentium — Pentium, P6 и Pentium IV с микроархитектурой NetBurst. К семейству

Pentium относятся упоминавшиеся ранее процессоры Pentium и Pentium MMX. История семейства P6 началась в 1995 году с появлением процессора Pentium Pro. Несмотря на схожие названия, внутренние архитектуры процессоров двух семейств были совершенно разными. Не вдаваясь в новые схемотехнические решения, реализованные в процессоре Pentium Pro, отметим его основные архитектурные свойства. Процессор поддерживал работу трех конвейеров, то есть мог обрабатывать до трех команд за один такт. Для этого использовались особые технологии обработки потока команд, которые подробнее будут рассмотрены в следующей главе. Процессор Pentium Pro использовал новую технологию работы с кэш-памятью. Наряду с уже обычным внутренним кэшем первого уровня размером 8 Кбайт в одном корпусе (но не на одной микросхеме) с процессором располагалась кэш-память второго уровня размером 256 Кбайт, для связи с которой имелась специальная 64-разрядная шина, работающая на частоте процессора. Шина данных процессора Pentium Pro имела разрядность 36 бита, что позволяло при определенных условиях организовать адресацию памяти до 64 Гбайт.

Процессор Pentium II, появившийся на свет в 1997 г., добавил к архитектуре процессора Pentium Pro поддержку MMX-команд. Кроме того, были увеличены размеры кэш-памяти всех уровней — кэш-память команд и данных первого уровня выросла до 16 Кбайт каждая, кэш-память второго уровня могла иметь величину 256, 512 Кбайт или 1 Гбайт. Кэш-память второго уровня могла работать на половине частоты работы процессора. Также процессор поддерживал множество технологий энергосбережения. Следующие две модели процессоров, выпущенные в 1998 г., — Celeron и Pentium II Xeon — были, соответственно, более «легкой» и более «тяжелой» модификациями процессора Pentium П. Celeron позиционировался как процессор для построения компьютерных систем массового использования. Pentium II Xeon предназначался для построения высокопроизводительных серверных систем.

Последний процессор семейства P6 — Pentium III, — выпущен в 2000 г. Его основное отличие — поддержка дополнительного набора MMX-команд, называемых SSE-расширением (SSE — Streaming SIMD Extensions) основного набора команд процессора. Для этого в архитектуру процессора был введен специальный блок.

На сегодняшний день последним 32-разрядным процессором является Pentium IV. Он позиционируется как процессор нового поколения с новым типом микроархитектуры, носящей название NetBurst. Подробнее основные архитектурные особенности процессора Pentium IV будут рассмотрены в следующей главе при обсуждении микроархитектуры NetBurst. Отметим лишь один важный в контексте нашего обсуждения момент — с появлением процессора Pentium IV система команд процессоров Intel пополнилась еще 144 новыми командами. В основном это команды для блока MMX с плавающей точкой, а также команды управления кэшированием и памятью. Условное название этой группы команд — SSE2 (Streaming SIMD Extensions 2).

Итоги

- Несмотря на значительные внешние различия структурно большинство современных компьютеров устроены примерно одинаково. В их состав обязательно входят центральный процессор, внешняя и оперативная память, устройства ввода-вывода и отображения информации.

- Работать компьютер заставляет некий «серый кардинал» — машинный язык. Пользователь может даже и не подозревать о его существовании. Общаться с компьютером пользователю помогают операционные системы, офисные пакеты, системы программирования и т. д. Современные технологии программирования позволяют создавать программы, не написав ни строчки кода. Но в «мозг» компьютера команды все же поступают на машинном языке.
- Машинный язык полностью отражает все архитектурные тонкости конкретного типа компьютеров. Следствием этого является его индивидуальность для каждого семейства ЭВМ. Чтобы эффективно использовать все возможности компьютера, применяют символический аналог машинного языка — язык ассемблера.
- Работать на компьютере можно и без знания ассемблера. Но элементом подготовки программиста-профессионала обязательно является изучение этого языка. Почему? Изучая ассемблер, человек обязательно попутно знакомится с архитектурой компьютера. А это, в свою очередь, позволяет ему в дальнейшем создавать более эффективные программы на других языках и объединять их при необходимости с программами на ассемблере.
- Процессоры Intel являются стандартом де-факто в своем сегменте рынка. За тридцать с лишним лет своего развития процессоры Intel из примитивных устройств, пригодных лишь для калькуляторов, превратились в сложные системы, содержащие более 7 000 000 транзисторов и поддерживающие работу высокопроизводительных приложений и многопроцессорных конфигураций.

Глава 2
Программно-аппаратная архитектура IA-32 процессоров Intel

- **Общее понятие об архитектуре ЭВМ**
- **Архитектура и свойства машины фон Неймана**
- **Общие и индивидуальные свойства процессоров Intel**
- **Архитектура IA-32 процессоров Intel**
- **Варианты микроархитектуры P6 (Pentium Pro/II/III) и NetBurst (Pentium IV)**
- **Программная модель архитектуры IA-32**
- **Режимы работы процессора архитектуры IA-32**
- **Набор регистров процессора архитектуры IA-32**
- **Организация памяти компьютера архитектуры IA-32**
- **Формирование физического адреса в реальном и защищенном режимах**

В главе 1 мы описали компьютер на «житейском» уровне. Его сердце — процессор. Без него компьютер представляет собой упорядоченный, но безжизненный набор аппаратных компонентов. Кроме различных схемотехнических решений, процессор живет благодаря машинному языку, «зашитому» в его устройстве микропрограммного управления. Именно машинный язык определяет логику работы процессора. В свою очередь, характеристики машинного языка полностью определяются особенностями того типа процессора, для которого этот язык предназначен.

При выборе конкретного типа (или модели) компьютера неизбежно возникают вопросы: как оценить его возможности, каковы его отличия от компьютеров других типов (моделей)? Рассмотрения одной лишь его функциональной схемы явно недостаточно, так как она вряд ли чем-то принципиально отличается от схем других машин. У всех компьютеров есть оперативная память, процессор, внешние устройства. В данном случае в содержании рекламы стоит обратить внимание на то, какие отличительные характеристики компьютера как товара продавец в первую очередь пытается довести до сведения потенциального покупателя. Обычно первые слова подобной рекламы: «Компьютер на базе процессора...». То есть все дело в процессоре. Именно он задает те правила, по которым в конечном итоге все устройства компьютера функционируют как единый механизм.

Для всего, что характеризует компьютер с точки зрения его функциональных программно-управляемых свойств, существует специальный широко распространенный термин — *архитектура* ЭВМ. Следует отметить, что большинство составляющих этот термин понятий относятся именно к процессору.

Архитектура ЭВМ

Однозначно определить понятие архитектуры ЭВМ довольно трудно, потому что при желании в него можно включить все, что связано с компьютерами вообще и какой-то конкретной моделью компьютера в частности. Попытаемся все же его формализовать.

Архитектура ЭВМ — это абстрактное представление ЭВМ, которое отражает ее структурную, схемотехническую и логическую организацию. Понятие архитектуры ЭВМ является комплексным, в него входят:

- структурная схема ЭВМ;
- средства и способы доступа к элементам структурной схемы ЭВМ;
- организация и разрядность интерфейсов ЭВМ;
- набор и доступность регистров;
- организация и способы адресации памяти;
- способы представления и форматы данных ЭВМ;
- набор машинных команд ЭВМ;
- форматы машинных команд;
- правила обработки нештатных ситуаций (прерываний).

Таким образом, описание архитектуры включает в себя практически всю необходимую для программиста информацию о компьютере. Понятие архитектуры ЭВМ — иерархическое. Допустимо вести речь как об архитектуре компьютера в целом, так и об архитектуре отдельных его составляющих, например, архитектуре процессора или архитектуре подсистемы ввода-вывода.

Все современные компьютеры обладают некоторыми общими и индивидуальными архитектурными свойствами. Индивидуальные свойства присущи только конкретной модели компьютера и отличают ее от своих больших и малых собратьев. Общие архитектурные свойства, наоборот, присущи некоторой, часто доволь-

но большой группе компьютеров. На сегодняшний день общие архитектурные свойства большинства современных компьютеров подпадают под понятие *фон-неймановской архитектуры.* Так названа архитектура по имени ученого фон Неймана. Когда фон Нейман начал заниматься компьютерами, программирование последних осуществлялось способом коммутирования. В первых ЭВМ для генерации нужных сигналов необходимо было с помощью переключателей выполнить ручное программирование всех логических схем. В первых машинах использовали десятичную логику, при которой каждый разряд представлялся десятичной цифрой и моделировался 10 электронными лампами. В зависимости от нужной цифры одна лампа включалась, остальные девять оставались выключенными. Фон Нейман предложил схему ЭВМ с программой в памяти и двоичной логикой вместо десятичной. Логически машину фон Неймана составляли пять блоков (рис. 2.1): оперативная память, арифметико-логическое устройство (АЛУ) с аккумулятором, блок управления, устройства ввода и вывода. Особо следует выделить роль аккумулятора. Физически он представляет собой регистр АЛУ. Для процессоров Intel, в которых большинство команд — двухоперандные, его роль не столь очевидна. Но существовали и существуют процессорные среды с однооперандными машинными командами. В них наличие аккумулятора играет ключевую роль, так как большинство команд используют его содержимое в качестве либо второго, либо единственного операнда команды.

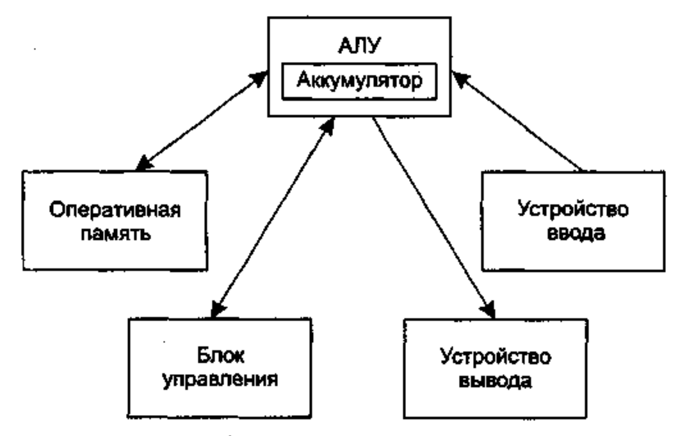


Рис. 2.1. Схема машины фон Неймана

Ниже описаны свойства и принципы работы машины фон Неймана.

- *Линейное пространство памяти.* Для оперативного хранения информации компьютер имеет совокупность ячеек с последовательной нумерацией (адресами) 0, 1, 2,... Данная совокупность ячеек называется *оперативной памятью.*
- *Принцип хранимой программы.* Согласно этому принципу, код программы и ее данные находятся в одном и том же адресном пространстве оперативной памяти.
- *Принцип микропрограммирования.* Суть этого принципа заключается в том, что машинный язык еще не является той конечной субстанцией, которая физически

приводит в действие процессы в машине. В состав процессора (см. главу 1) входит *устройство микропрограммного управления*, поддерживающее набор действий-сигналов, которые нужно сгенерировать для физического выполнения каждой машинной команды.

- *Последовательное выполнение программ.* Процессор выбирает из памяти команды строго последовательно. Для изменения прямолинейного хода выполнения программы или осуществления ветвления необходимо использовать специальные команды. Они называются командами *условного* и *безусловного* переходов.
- *Отсутствие разницы между данными и командами в памяти.* С точки зрения процессора, нет принципиальной разницы между данными и командами. Данные и машинные команды находятся в одном пространстве памяти в виде последовательности нулей и единиц. Это свойство связано с предыдущим. Процессор, поочередно обрабатывая некоторые ячейки памяти, всегда пытается трактовать содержимое ячеек как коды машинных команд, а если это не так, то происходит аварийное завершение программы. Поэтому важно всегда четко разделять в программе пространства данных и команд.
- *Безразличие к назначению данных.* Машине все равно, какую логическую нагрузку несут обрабатываемые ею данные.

Данный учебник посвящен вопросам программирования процессоров фирмы Intel и Intel-совместимых процессоров других фирм. Поэтому в качестве примера *индивидуальных* архитектурных принципов компьютера, в силу иерархичности этого понятия, рассмотрим индивидуальные архитектурные принципы и свойства процессоров Intel. Их полное рассмотрение не является нашей целью, уделим внимание лишь тем их них, которые наиболее характерны и понадобятся нам для дальнейшего изложения. Более подробно вопросы организации архитектуры компьютера описаны в [7].

Согласно материалам фирмы Intel, индивидуальные архитектурные свойства и принципы работы всех ее процессоров, начиная с i8086 и заканчивая Pentium IV, выстроены в рамках единой архитектуры, которая позже получила название IA-32 (32-bit Intel Architecture). Эта архитектура не является «закостенелым» образованием. В процессе эволюции процессоров Intel она постоянно изменяется и развивается. Каждый из процессоров вносил в IA-32 одно или несколько архитектурных новшеств. Несмотря на то, что датой рождения архитектуры IA-32 нужно считать дату появления на свет процессора i80386, предыдущие модели процессоров также внесли существенный вклад в представление ее принципов и свойств. Если вернуться к материалу главы 1, то можно проследить за вкладом, который внес каждый из процессоров Intel в ходе своей эволюции в формирование элементов архитектуры IA-32. Так, благодаря процессорам i8086/88 в IA-32 существует сегментация памяти, i80286 ввел защищенный режим и т. д.

Архитектура IA-32

В рамках архитектуры IA-32 существует и развивается ряд микроархитектур. Ранее было показано, что понятие архитектуры компьютера иерархично и что существуют общие и индивидуальные свойства архитектуры. В контексте обсуждения

архитектуры процессоров Intel имеет смысл также позиционировать различные их архитектурные свойства и принципы как общие и индивидуальные. К *индивидуальным* архитектурным свойствам и принципам можно отнести существующие в рамках различных микроархитектур (см. далее). Что касается *общих* архитектурных свойств и принципов IA-32, то к ним относятся те, которые имеют место для всех процессоров Intel или, по крайней мере, существуют вне рамок конкретной микроархитектуры для большого числа моделей процессоров. Так как процессор в основном определяет логику работы компьютера, то и названия большинства общих архитектурных свойств и принципов IA-32 совпадают с названиями аналогичных свойств и принципов компьютера: номенклатура программно-доступных регистров; организация и способы адресации памяти; номенклатура режимов работы процессоров; организация и разрядность внешних интерфейсов ЭВМ; способы представления и форматы данных; набор и форматы машинных команд ЭВМ; порядок обработки прерываний. Практически все эти общие архитектурные свойства и принципы составляют программную модель процессора, которая будет рассмотрена в дальнейшем.

Варианты микроархитектуры процессоров Intel

Понятие микроархитектуры впервые было определено Intel для процессоров семейства Pentium Pro. Его введение объяснялось необходимостью правильного позиционирования новых процессоров среди существующих. Внешняя программная модель (логическая) 32-разрядных процессоров изменялась только в сторону развития, в то время как их исполнительная (физическая) часть могла быть совершено разной. Понятие микроархитектуры ориентировано на описание особенностей исполнительной части процессоров, то есть того, какими способами и какими средствами процессор выполняет обработку машинного кода (рис. 2.2). На сегодняшний день в рамках IA-32 существует две микроархитектуры процессоров Intel: P6 и NetBurst.

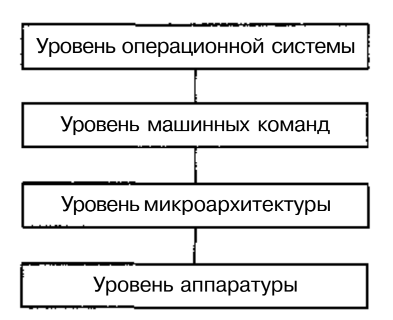


Рис. 2.2. Представление компьютера в виде уровней

Микроархитектура P6

Микроархитектуру P6 поддерживают такие процессоры Intel, как Pentium Pro, Pentium II (Xeon), Celeron, Pentium III (Xeon). Эта микроархитектура является, по определению Intel, *трехходовой* (three-way) *суперскалярной конвейерной* архи-

тектурой. Термин «трехходовая» означает поддержку технологий параллельного вычисления, позволяющих процессору одновременно (за один такт) обрабатывать до трех инструкций. Проблема оптимальной обработки потока машинных команд является ключевой при разработке любого процессора. Поэтому для большей ясности необходимо показать эту проблему в развитии. В компьютере фон-неймановской архитектуры существуют две основные стадии исполнения команды — выборка очередной команды из памяти и собственно ее исполнение. В первых процессорах Intel все блоки процессора работали последовательно, начиная с этапа выборки очередной команды из памяти и заканчивая этапом завершения ее обработки процессором. Напоминание об этом осталось в названии регистра IP/EIP — (Instruction Pointer — указатель инструкции). До появления процессоров Intel с конвейерной архитектурой данный регистр непосредственно указывал на очередную команду, подлежащую выполнению. Процессоры Intel относятся к группе CISC-процессоров, в которых для выполнения одной команды может требоваться от единиц до нескольких десятков процессорных тактов. При такой обработке команд увеличение производительности может быть достигнуто только повышением частоты генерации машинных тактов. Простое увеличение частоты работы процессора не имеет смысла, так как есть физически обусловленная верхняя граница, до которой ее можно поднимать. По этому пути разработчики Intel шли до процессора i80386 включительно. В ходе исполнения команды есть и другое узкое место — выборка команды из памяти. Это затратная по времени операция. Частичное решение проблемы было найдено еще на заре развития компьютерной техники в виде буфера упреждающей выборки. Развитием этой и реализацией других идей стал *конвейер* — специальное устройство, существующее на уровне архитектуры исполнительной части компьютера. Благодаря конвейеру исполнение команды разбивается на несколько стадий, каждая из которых реализует некоторую элементарную операцию общего процесса обработки команды. Впервые для процессоров Intel конвейер был реализован в архитектуре процессора i80486. Конвейер i80486 имеет пять ступеней, которые соответствуют перечисленным далее стадиям обработки машинной команды.

1. Выборка команды из кэш-памяти или из оперативной памяти.
2. Декодирование команды.
3. Генерация адреса, в ходе которой определяются адреса операндов в памяти и выполняется выборка операндов.
4. Выполнение операции с помощью АЛУ.
5. Запись результата (место записи результата зависит от алгоритма работы конкретной машинной команды).

В чем преимущество такого подхода? Очередная команда после ее выборки попадает в блок декодирования. Таким образом блок выборки освобождается и может выбрать следующую команду. В результате на конвейере могут находиться в различной стадии выполнения пять команд. Скорость вычисления в результате существенно возрастает.

В процессорах Pentium конвейерная архитектура была усовершенствована и получила название *суперскалярной*. В отличие от *скалярной* архитектуры i80486

(с одним конвейером), первые модели процессоров Pentium имели два конвейера. В идеале такой суперскалярный процессор должен выполнять две команды за машинный такт. Но не все так просто. Реально два конвейера Pentium не были функционально равнозначными. В связи с этим они даже имели разные названия — u-конвейер (главный) и v-конвейер (второстепенный). Главный конвейер был полнофункциональным и мог выполнять любые машинные команды. Функциональность второстепенного конвейера была ограничена основными целочисленными командами и одной командой с плавающей точкой (FXCH). Внутренняя структура обоих конвейеров такая же, как у i80486 с одним общим блоком выборки команд. Для того чтобы два разных по функциональным возможностям конвейера могли обеспечить предельную эффективность (две выполненных команды за такт работы процессора), необходимо было группировать команды из входного потока в совместимые пары. Важно заметить, что исходная последовательность команд входного потока была неизменной. Если процессору не удавалось собрать совместимую пару, то выполнялась одна команда на u-конвейере. Оставшуюся команду процессор пытался «спарить» со следующей командой входного потока.

Вернемся к процессорам микроархитектуры P6. Они имеют другую структуру конвейера. Собственно конвейера в понимании i80486 и первых Pentium уже нет. Конвейеризация заключается в том, что весь процесс обработки команд разбит на 12 стадий, которые исполняются различными блоками процессора. Сколько именно команд обрабатывается процессором, сказать трудно. Термин *трехходовой* означает лишь то, что для исполнения из входного потока выбираются до трех команд. Известен верхний предел — в процессоре в каждый момент времени могут находиться до 30 команд в различной стадии исполнения. Детали этого процесса скрыты за понятием *динамическое исполнение с нарушением исходного порядка следования машинных команд* (out of order), что означает исполнение команд в порядке, определяемом исполнительным устройством процессора, а не исходной последовательностью команд. В основу технологии динамического исполнения положены три концепции:

- *Предсказание правильного адреса перехода.* Основная задача механизма предсказания — исключить перезагрузку конвейера. Под *переходом* понимается запланированное алгоритмом изменение последовательного характера выполнения программы. Как показывает статистика, типичная программа на каждые 6-8 команд содержит 1 команду перехода. Последствия обработки перехода предсказать несложно: при наличии конвейера через каждые 6-8 команд его нужно очищать и заполнять заново в соответствии с адресом перехода. Все преимущества конвейеризации теряются. Поэтому в архитектуру Pentium в состав устройства выборки/декодирования (см. главу 1) был введен *блок предсказания переходов.* Вероятность правильного предсказания составляет около 80 %.
- *Динамический анализ потока данных.* Анализ проводится с целью определения зависимостей команд программы от данных и регистров процессора с последующей оптимизацией выполнения потока команд. Главный критерий здесь — максимально полная загрузка процессора. Для реализации данного критерия допускается нарушение исходного порядка следования команд. Сбоя при этом не7 происходит, так как внешне логика работы программы не нарушается. 39999-

Подобная внутренняя неупорядоченность исполнения команд позволяет держать процессор загруженным даже тогда, когда данные в кэш-памяти второго уровня отсутствуют и необходимо тратить время на обращение за ними в оперативную память.

- *Спекулятивное исполнение* — способность процессора исполнять машинные команды на участках программы с условными переходами и циклами до того, как эти переходы будут разрешены алгоритмом программы. Если переход предсказан правильно, то процессор к этому моменту уже имеет исполненный код, в противном случае весь конвейер нужно очищать, загружать и исполнять код новой ветви программы, что очень накладно.

Рассмотрим порядок функционирования исполнительного устройства микроархитектуры Р6 и реализацию с его помощью описанных ранее технологий. Это рассмотрение не является строгим, кое-где для лучшего понимания оно упрощено. Для иллюстрации будем использовать схему, представленную на рис. 2.3 и являющуюся развитием схемы на рис. 1.2 (см. главу 1). Из схемы видно, что структурно процессор микроархитектуры Р6 состоит из нескольких подсистем.

- *Подсистема памяти* состоит из системной шины, кэша второго уровня L2, устройства шинного интерфейса, кэша первого уровня L1 (инструкций и данных), устройства связи с памятью и буфера переупорядочивания запросов к памяти.
- *Устройство выборки/декодирования* состоит из устройства выборки команд, блока предсказания переходов, в который входят блоки меток перехода и вычисления адреса следующей инструкции, устройства декодирования, устройства микропрограммного управления и таблицы псевдонимов регистров.
- *Буфер команд.*
- *Устройство диспетчеризации/исполнения* содержит буфер микроопераций, готовых к исполнению, и пять исполнительных устройств (два — для исполнения целочисленных операций, два — для исполнения операций с плавающей точкой, а также устройство связи с памятью). Необходимо заметить, что здесь допущена вольность в трактовке назначения исполнительных устройств: выделены устройства для выполнения обычных команд (целочисленных и с плавающей точкой) и MMX-команд (также целочисленных и с плавающей точкой). Реальное деление несколько иное. Такое допущение сделано исключительно с учебной целью — для более осознанного перехода от архитектуры к системе команд ассемблера.
- *Блок удаления и восстановления.*

Рассмотрим подсистему памяти. Для бесперебойной работы процессора в микроархитектуре Р6 используется два уровня кэш-памяти[1]. Кэш-память первого уровня состоит из кэшей команд и данных размером по 8 Кбайт, расположенных внутри процессора в непосредственной близости к его исполнительной части. Кэш-память второго уровня является внешней по отношению к процессору (но в едином кон-

[1] Кэширование — способ увеличения быстродействия системы за счет хранения часто используемых данных и кодов в так называемой кэш-памяти, находящейся внутри процессора (кэш-память первого уровня) либо в непосредственной близости от него (кэш-память второго уровня).

2 Зак. 256

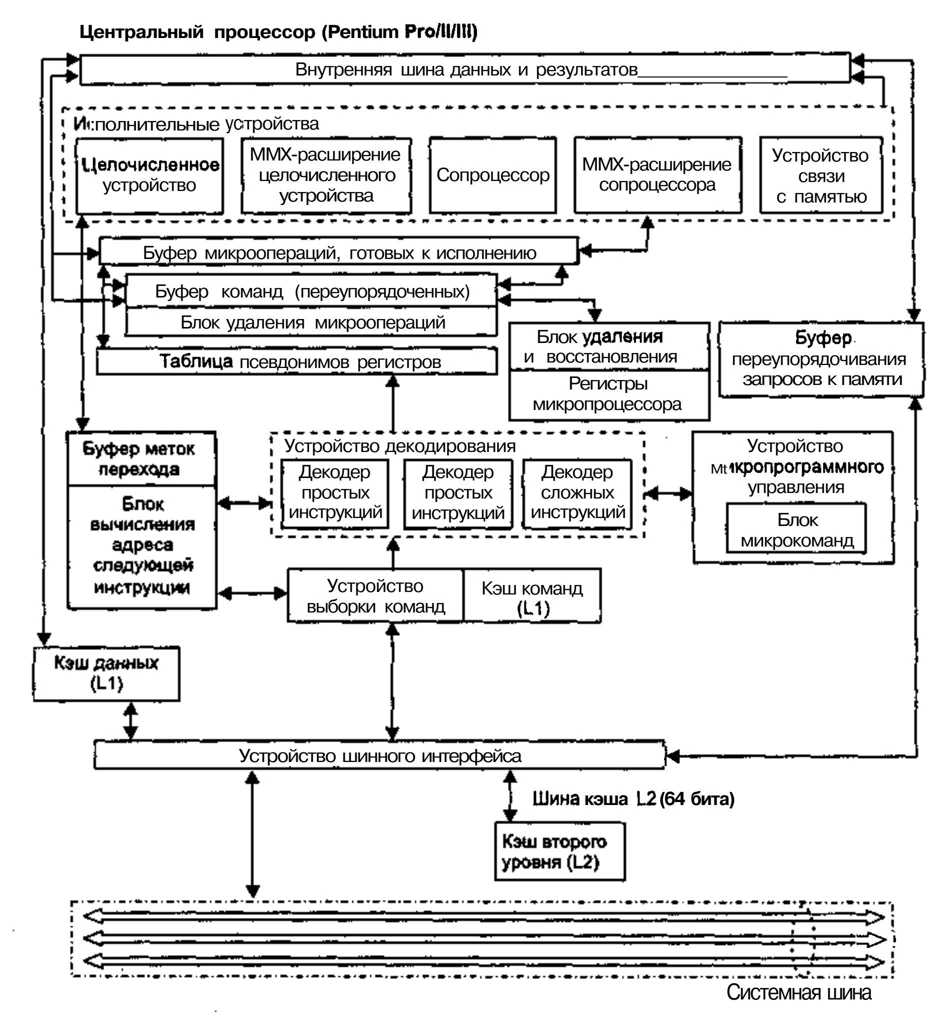


Рис. 2.3. Структурная схема процессора семейства P6 (Pentium Pro/11/III)

структиве с ним), имеет значительно б ольший размер (256 Кбайт, 512 Кбайт или 1 Мбайт) и соединена с ядром процессора посредством 64-разрядной шины. Разделение кэш-памяти на две части (для кода и данных) обеспечивает бесперебойную поставку машинных инструкций и элементов данных в исполнительное устройство процессора. Исходные данные для кэш-памяти первого уровня предоставляет кэш-память второго уровня. Заметьте, что информация из нее поступает на устройство шинного интерфейса и далее в соответствующую кэш-память первого уровня по 64-разрядной шине. При этом благодаря более быстрому обновлению содержимого кэш-памяти первого уровня обеспечивается высокий темп работы процессора.

Устройство шинного интерфейса обращается к оперативной памяти системы через внешнюю системную шину. Эта 64-разрядная шина ориентирована на обработку запросов, то есть каждый шинный запрос обрабатывается отдельно и требу-

ет обратной реакции. Пока устройство шинного интерфейса ожидает ответа на один запрос шины, возможно формирование многочисленных дополнительных запросов. Все они обслуживаются в порядке поступления. Считываемые по запросу данные помещаются в кэш второго уровня. То есть процессор посредством устройства шинного интерфейса читает команды и данные из кэша второго уровня. Устройство шинного интерфейса взаимодействует с кэшем второго уровня через 64-разрядную шину кэша, которая также ориентирована на обработку запросов и функционирует на тактовой частоте процессора. Доступ к кэшу первого уровня осуществляется через внутренние шины на тактовой частоте процессора. Синхронная работа с системной памятью кэш-памяти обоих уровней осуществляется благодаря специальному протоколу MESI [7], [13].

Запросы от команд на получение операндов из памяти в исполнительном устройстве процессора обслуживаются посредством *устройства связи с памятью* и *буфера переупорядочивания запросов к памяти.* Эти два устройства специально включены в схему для того, чтобы обеспечить бесперебойное снабжение исполняемых команд необходимыми данными. Особо стоит подчеркнуть роль буфера переупорядочивания запросов к памяти. Он отслеживает все запросы к операндам в памяти и выполняет функции планирующего устройства. Если нужные для очередной операции данные в кэш-памяти первого уровня (L1) отсутствуют, то буфер переупорядочивания запросов к памяти автоматически передает информацию о неудачном обращении к данным кэшу второго уровня (L2). Если и в кэше L2 нужных данных не оказалось, то буфер переупорядочивания запросов к памяти заставляет устройство шинного интерфейса сформировать запрос к оперативной памяти компьютера.

Устройство выборки/декодирования извлекает одну 32-байтную строку кэша команд (L1) за такт и передает в декодер, который преобразует ее в последовательность микроопераций. Поток микроопераций (пока он еще соответствует последовательности исходных команд) поступает в *буфер команд. Устройство выборки команд* вычисляет указатель на следующую команду, подлежащую выборке, на основании информации трех источников: буфера меток перехода, состояния прерывания/исключения и сообщения от исполнительного целочисленного устройства об ошибке в предсказании метки перехода. Важная часть этого процесса — предсказание метки перехода, которое выполняется по специальному алгоритму. В его основе лежит работа с *буфером меток перехода,* который содержит информацию о последних 256 переходах. Если очередная команда, выбираемая из памяти, является командой перехода, то содержащийся в ней адрес перехода сравнивается с адресами, уже находящимися в буфере меток перехода. Если этот адрес уже есть в буфере меток переходов, то он станет адресом следующей команды, с которой устройство выборки будет извлекать очередную команду. Если искомого адреса перехода в буфере нет, то выборка команд из памяти будет продолжена до момента исполнения команды перехода исполнительным устройством. В результате ее исполнения становится ясно, было ли правильным решение об адресе начала выборки следующих команд после выборки команды перехода. Если предсказанный переход оказывается неверным, то конвейер сбрасывается и загружается заново в соответствии с адресом перехода. Цель предсказания переходов — в том, чтобы

устройство исполнения постоянно было занято полезной работой и сброс конвейера производился как можно реже.

Устройство выборки команд выбирает команды для исполнения и помещает их в устройство декодирования. Устройство декодирования состоит из трех параллельно работающих *декодеров* (два простых и один сложный). Именно эти декодеры воплощают в жизнь понятие исполнения с нарушением исходного порядка следования команд (out of order) и являются теми самыми тремя входами (three way) в исполнительное устройство процессора. Декодеры преобразуют команды процессора в микрооперации. Микрооперации представляют собой примитивные команды, которые выполняются пятью *исполнительными устройствами* процессора, работающими параллельно. Многие машинные команды преобразуются в одиночные микрооперации (это делает простой декодер), а некоторые машинные команды — в последовательность от двух и более (оптимально — четырех) микроопераций (это делает сложный декодер). Информация о последовательности микроопераций для реализации конкретной машинной команды содержится в *устройстве микропрограммного управления*. Кроме команд, декодеры обрабатывают также префиксы команд. Декодер команд может формировать до шести микроопераций за такт — по одной от простых декодеров и до четырех от сложного декодера. Для достижения наибольшей производительности работы декодеров необходимо, чтобы на их вход поступали команды, которые декодируются шестью микрооперациями в последовательности 4 + 1 + 1. Если время работы программы критично, то имеет смысл провести ее оптимизацию, заключающуюся в переупорядочивании исходного набора команд таким образом, чтобы группы команд формировали последовательности микроопераций по схеме 4 + 1 + 1. После того как команды разбиты на микрооперации, порядок их выполнения трудно предсказать. При этом могут возникнуть проблемы с таким критичным ресурсом, как регистры. Суть здесь в том, что если в двух соседних фрагментах программы данные помещались в одинаковые регистры, откуда они, возможно, записывались в некоторые области памяти, а после переупорядочивания эти фрагменты перемешались, то как разобраться в том, какие регистры и где использовались. Эта проблема носит название *проблемы ложных взаимозависимостей* и решается с помощью *механизма переименования регистров*. Основу этого механизма составляет набор из 40 внутренних универсальных регистров, которые и задействуются в реальных вычислениях исполнительным устройством абсолютно прозрачно для программ. Универсальные регистры могут работать как с целыми числами, так и со значениями с плавающей точкой. Информация о действительных именах регистров процессора и их внутренних именах (номерах универсальных регистров) помещается в *таблицу псевдонимов регистров*.

В заключение процесса декодирования *устройство управления таблицей псевдонимов регистров* добавляет к микрооперациям биты состояния и флаги, чтобы подготовить их к неупорядоченному выполнению, после чего посылает получившиеся микрооперации в *буфер переупорядоченных команд*. Нужно заметить, что новый порядок их следования не соответствует порядку следования соответствующих команд в исходной программе. Буфер переупорядоченных команд представляет собой массив ассоциативной памяти, физически выполненный в виде 40 регистров и представляющий собой кольцевую структуру, элементы которой содержат

два типа микроопераций: ожидающие своей очереди на исполнение и уже частично выполненные, но не до конца из-за их переупорядочивания и зависимости от других частично или полностью не выполненных микроопераций. Устройство диспетчеризации/исполнения может выбирать микрооперации из этого буфера в любом порядке.

Устройство диспетчеризации/исполнения планирует и исполняет неупорядоченную последовательность микроопераций из буфера переупорядоченных команд. Но оно не занимается непосредственной выборкой микроопераций из буфера переупорядоченных команд, так как в нем могут содержаться и не готовые к исполнению микрооперации. Этим занимается устройство, управляющее специальным буфером, который условно назовем *буфером микроопераций, готовых к исполнению.* Оно постоянно сканирует буфер переупорядоченных команд в поисках микроопераций, готовых к исполнению (фактически это означает доступность всех операндов), после чего посылает их соответствующим исполнительным устройствам, если они не заняты. Результаты исполнения микроопераций возвращаются в буфер переупорядоченных команд и сохраняются там наряду с другими микрооперациями до тех пор, пока не будут удалены *устройством удаления и восстановления.*

Подобная схема планирования и исполнения программ реализует классический принцип неупорядоченного выполнения, при котором микрооперации посылаются исполнительным устройствам вне зависимости от их расположения в исходном алгоритме. В случае, если к выполнению одновременно готовы две или более микрооперации одного типа (например, целочисленные), то они выполняются в соответствии с принципом FIFO (First In, First Out — первым пришел, первым ушел), то есть в порядке поступления в буфер переупорядоченных команд.

Непосредственно исполнительное устройство состоит из пяти блоков, каждый из которых обрабатывает свой тип микроопераций: два целочисленных устройства, два устройства для вычислений с плавающей точкой и одно устройство связи с памятью. Таким образом, за один машинный такт одновременно исполняется пять микроопераций.

Два целочисленных исполнительных устройства могут параллельно обрабатывать две целочисленные микрооперации. Одно из этих целочисленных исполнительных устройств специально предназначено для работы с микрооперациями переходов. Оно способно обнаружить непредсказанный переход и сообщить об этом устройству выборки команд, чтобы перезапустить конвейер. Такая операция реализована следующим образом. Декодер команд отмечает каждую микрооперацию перехода и адрес перехода. Когда целочисленное исполнительное устройство выполняет микрооперацию перехода, то оно определяет, был предсказан переход или нет. Если переход предсказан правильно, то микрооперация отмечается пригодной для использования, и выполнение продолжается по предсказанной ветви. Если переход предсказан неправильно, то целочисленное исполнительное устройство изменяет состояние всех последующих микроопераций с тем, чтобы удалить их из буфера переупорядоченных команд. После этого целочисленное устройство помещает метку перехода в буфер меток перехода, который, в свою очередь, совместно с устройством выборки команд перезапускает конвейер относительно нового исполнительного адреса.

Устройство связи с памятью управляет загрузкой и сохранением данных для микроопераций. Для их загрузки в исполнительное устройство достаточно определить только адрес памяти, поэтому такое действие кодируется одной микрооперацией. Для сохранения данных необходимо определять и адрес, и записываемые данные, поэтому это действие кодируется двумя микрооперациями. Та часть устройства связи с памятью, которая управляет сохранением данных, имеет два блока, позволяющие ему обрабатывать адрес и данные для микрооперации параллельно. Это позволяет устройству связи с памятью выполнить загрузку и сохранение данных для микроопераций параллельно в одном такте.

Исполнительные устройства с плавающей точкой аналогичны устройствам в более ранних моделях процессора Pentium. Было добавлено только несколько новых команд с плавающей точкой для организации условных переходов и перемещений.

Последний блок в этой схеме выполнения команд исходной программы — *блок удаления и восстановления,* задачей которого является возврат вычислительного процесса в рамки, определенные исходной последовательностью команд. Для этого он постоянно сканирует буфер переупорядоченных команд на предмет обнаружения полностью выполненных микроопераций, не имеющих связи с другими микрооперациями. Такие микрооперации удаляются из буфера переупорядоченных команд и восстанавливаются в порядке, соответствующем порядку следования команд исходной программы с учетом прерываний, исключений, точек прерывания и переходов. Блок удаления и восстановления может удалить три микрооперации за один машинный такт. При восстановлении порядка следования команд блок удаления и восстановления записывает результаты в реальные регистры процессора и в оперативную память.

Таким образом, система динамического исполнения команд позволяет организовать прохождение команд программы через исполнительное устройство процессора эффективнее, чем это было в конвейере процессора i80486 и первых процессоров Pentium.

Микроархитектура NetBurst

Микроархитектура NetBurst, реализованная в процессоре Pentium IV, является развитием идей микроархитектуры Р6, поэтому рассматривается довольно конспективно. Судя по названию (net — сеть, burst — прорыв), микроархитектура NetBurst призвана обеспечить некий «сетевой прорыв». Очевидно, что этим разработчики хотели подчеркнуть те новые особенности процессора Pentium IV, которые позволяют организовать более быструю и эффективную работу приложений в современных сетевых и мультимедийных информационных средах. Отметим наиболее важные свойства новой микроархитектуры.

- *Быстрая исполнительная часть процессора.* АЛУ процессора работает на удвоенной частоте процессора. За каждый такт процессора выполняются две основные целочисленные команды. Обеспечена более высокая пропускная способность потока команд через исполнительную часть процессора и уменьшены различные задержки.
- *Гиперконвейерная технология.* Гиперконвейер Pentium IV состоит из 20 ступеней. Цель увеличения длины конвейера — упрощение задач, реализуемых каж-

дой из его ступеней, и, как следствие, упрощение соответствующей аппаратной логики.

- *Улучшенная технология динамического исполнения благодаря более глубокой «произвольности» в порядке исполнения кода и усовершенствованной системе предсказания переходов.* Размер буфера меток перехода увеличен до 4 Кбайт (Pentium III — 512 байт). Усовершенствован и сам алгоритм предсказания. В результате вероятность предсказания перехода возрастает до 95 %.
- *Новая подсистема кэширования.* Отсутствует кэш команд первого уровня. Вместо него введен кэш трасс. *Трассами* называются последовательности микроопераций, в которые были декодированы ранее выбранные команды. Кэш трасс способен хранить до 12 Кбайт микроопераций и доставлять исполнительному ядру до 3 микроопераций за такт. Кэш второго уровня работает на полной частоте ядра процессора.

Структурная схема процессора Pentium IV показана на рис. 2.4.

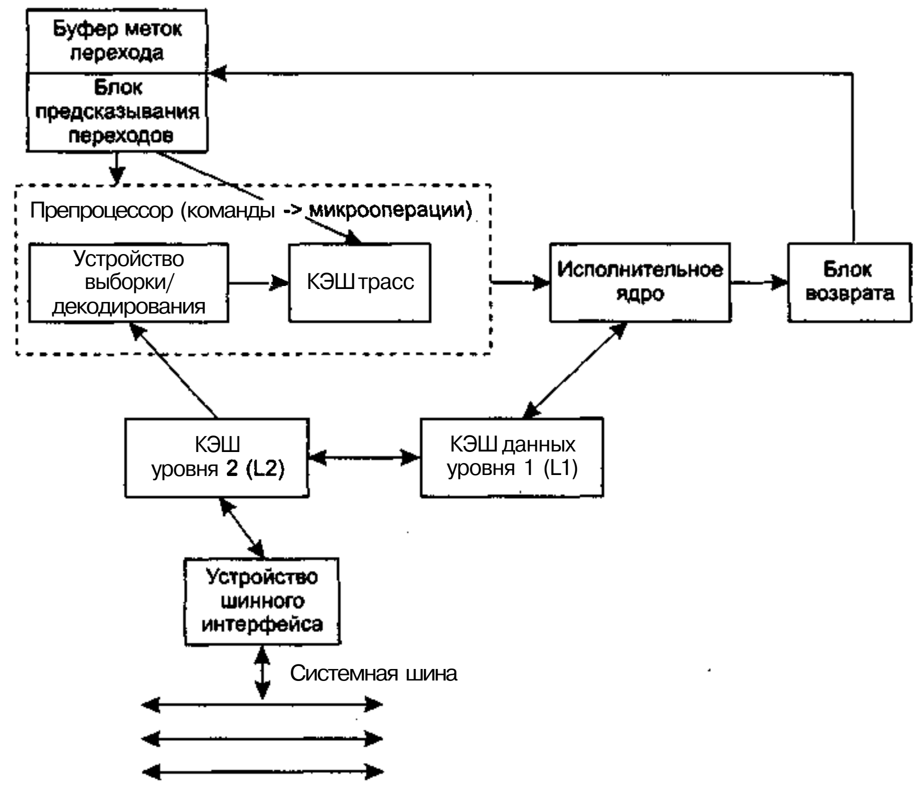


Рис. 2.4. Структурная схема процессора Pentium IV

Микроархитектура NetBurst поддерживает еще одну новую технологию — HyperThreading. Данная технология позволяет на базе одного физического процессора Pentium IV моделировать несколько логических, каждый из которых имеет собственное архитектурное пространство IA-32. Под архитектурным пространством

IA-32 понимается совокупность регистров данных, сегментных регистров, системных регистров и регистров MSR. Каждый логический процессор имеет также собственный контроллер прерываний APIC.

На этом, наверное, следует завершить обсуждение общих вопросов, связанных с архитектурой процессоров Intel. Более подробную информацию по этой теме, в том числе о логике функционирования и характеристиках микроархитектур P6 и NetBurst, можно получить из материала книги [13]. Наша ближайшая задача — разобраться, как управлять этими сложными процессорами. Программисту совершенно необязательно помнить о схемотехнических тонкостях реализации процессора, ему достаточно знать, какие части процессора ему доступны и как с ними взаимодействовать для реализации некоторой задачи. Первый шаг в этом направлении — знакомство с *программной моделью процессора.* Эта модель описывает видимые для программиста объекты архитектуры процессора, знание которых позволяет программисту эффективно и в полном объеме использовать возможности процессора.

Программная модель IA-32

Любая выполняющаяся программа получает в свое распоряжение определенный набор ресурсов процессора. Эти ресурсы необходимы для обработки и хранения в памяти команд и данных программы, а также информации о текущем состоянии программы и процессора. Программную модель процессора в архитектуре IA-32 процессоров Intel составляет следующий набор ресурсов (рис. 2.5):

- пространство адресуемой памяти до 2^{32} - 1 байт (4 Гбайт), для Pentium III/IV — до 2^{36} - 1 байт (64 Гбайт);
- набор регистров для хранения данных общего назначения;
- набор сегментных регистров;
- набор регистров состояния и управления;
- набор регистров устройства вычислений с плавающей точкой (сопроцессора);
- набор регистров целочисленного MMX-расширения, отображенных на регистры сопроцессора (впервые появились в архитектуре процессора Pentium MMX);
- набор регистров MMX-расширения с плавающей точкой (впервые появились в архитектуре процессора Pentium III);
- программный стек — специальная информационная структура, работа с которой предусмотрена на уровне машинных команд (более подробно она будет обсуждена позже).

Это основной набор ресурсов. Кроме того, к ресурсам, поддерживаемым архитектурой IA-32, необходимо отнести порты ввода-вывода, счетчики мониторинга производительности.

Программные модели более ранних процессоров (i486, первые Pentium) отличаются меньшим размером адресуемого пространства оперативной памяти (2^{32} - 1, так как разрядность их шины адреса составляет 32 бита) и отсутствием некоторых групп регистров. Для каждой группы регистров в скобках показано, начиная

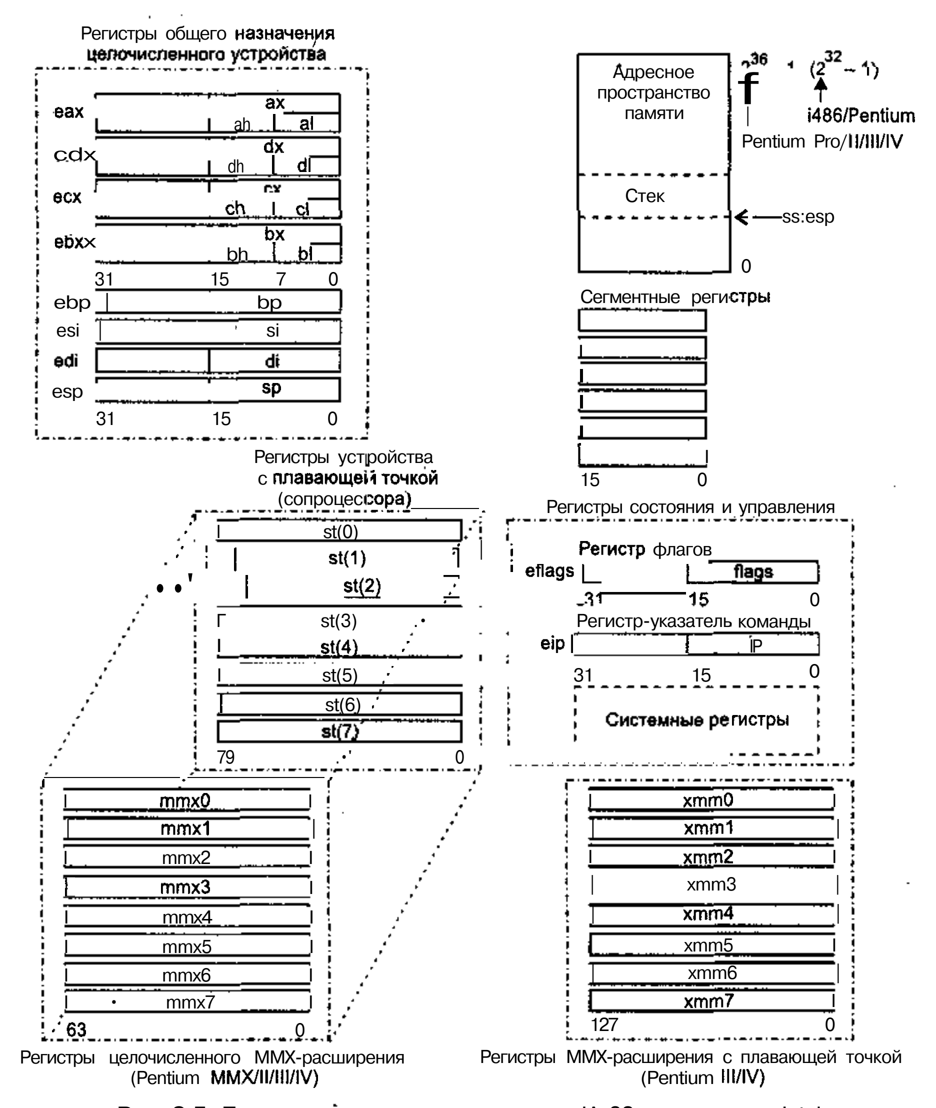


Рис. 2.5. Программная модель архитектуры IA-32 процессоров Intel

с какой модели данная группа регистров появилась в программной модели процессоров Intel. Если такого обозначения нет, то это означает, что данная группа регистров присутствовала в процессорах i386 и i486. Что касается еще более ранних процессоров i8086/88, то на самом деле они тоже полностью представлены на схеме, но составляют лишь ее небольшую ее часть. В программную модель данных процессоров входят 8- и 16-разрядные регистры общего назначения, сегментные регистры, регистры FLAGS, IP и адресное пространство памяти размером до 1 Мбайт.

Свойства некоторых перечисленных далее программно-доступных ресурсов определяются текущим режимом работы процессора.

Режимы работы процессора архитектуры IA-32

Режим работы процессора определяет поведение, номенклатуру и свойства доступных ресурсов процессора. Перевод процессора из одного режима в другой осуществляется специальными программными и аппаратными методами.

В рамках архитектуры IA-32 доступны следующие режимы работы процессора.

- *Режим реальных адресов,* или просто *реальный режим* (real mode) — это режим, в котором работал i8086. Наличие его в i486 и Pentium обусловлено тем, что фирма Intel старается обеспечить в новых моделях процессоров возможность функционирования программ, разработанных для ранних моделей.
- *Защищенный режим* (protected mode) позволяет максимально реализовать все идеи, заложенные в процессорах архитектуры IA-32, начиная с i80286. Программы, разработанные для i8086 (реального режима), не могут функционировать в защищенном режиме. Одна из причин этого связана с особенностями формирования физического адреса в защищенном режиме.
- *Режим виртуального процессора 8086* предназначен для организации многозадачной работы программ, разработанных для реального режима (процессора i8086), совместно с программами защищенного режима. Переход в этот режим (virtual 8086 mode) возможен, если процессор уже находится в защищенном режиме. Работа программ реального режима в режиме виртуального i8086 возможна благодаря тому, что процесс формирования физического адреса для них производится по правилам реального режима.
- *Режим системного управления* (System Management Mode, SMM) — это новый режим работы процессора, впервые появившийся в процессоре Pentium. Он обеспечивает операционную систему механизмом для выполнения машинно-зависимых функций, таких как перевод компьютера в режим пониженного энергопотребления или выполнения действий по защите системы. Для перехода в данный режим процессор должен получить специальный сигнал SMI от усовершенствованного программируемого контроллера прерываний (Advanced Programmable Interrupt Controller, APIC), при этом сохраняется состояние вычислительной среды процессора. Функционирование процессора в этом режиме подобно его работе в режиме реальных адресов. Возврат из этого режима производится специальной командой процессора.

Процессор всегда начинает работу в реальном режиме.

Набор регистров

Регистрами называются области высокоскоростной памяти, расположенные внутри процессора в непосредственной близости от его исполнительного ядра. Доступ к ним осуществляется несравнимо быстрее, чем к ячейкам оперативной памяти. Соответственно, машинные команды с операндами в регистрах выполняются максимально быстро, поэтому в программах на языке ассемблера регистры используются очень интенсивно. К сожалению, архитектура IA-32 предоставляет в распоряжение программиста не слишком много регистров, поэтому они являются критически важным ресурсом и за их содержимым приходится следить очень внимательно.

Большинство регистров имеют определенное функциональное назначение. С точки зрения программиста, их можно разделить на две большие группы.

- Первую группу образуют пользовательские регистры, к которым относятся:
  - регистры общего назначения EAX/AX/AH/AL, EBX/BX/BH/BL, EDX/DX/DH/DL, ECX/CX/CH/CL, EBP/BP, ESI/SI, EDI/DI, ESP/SP предназначены для хранения данных и адресов, программист может их использовать (с определенными ограничениями) для реализации своих алгоритмов;
  - сегментные регистры CS, DS, SS, ES, FS, GS используются для хранения адресов сегментов в памяти;
  - регистры сопроцессора ST(0), ST(1), ST(2), ST(3), ST(4), ST(5), ST(6), ST(7) предназначены для написания программ, использующих тип данных с плавающей точкой (глава 17);
  - целочисленные регистры MMX-расширения MMX0, MMX1, MMX2, MMX3, MMX4, MMX5, MMX6, MMX7;
  - регистры MMX-расширения с плавающей точкой XMM0, XMM1, XMM2, XMM3, XMM4, XMM5, XMM6, XMM7;
  - регистры состояния и управления (регистр флагов EFLAGS/FLAGS и регистр-указатель команды EIP/IP) содержат информацию о состоянии процессора, исполняемой программы и позволяют изменить это состояние.
- Во вторую группу входят системные регистры, то есть регистры, предназначенные для поддержания различных режимов работы, сервисных функций, а также регистры, специфичные для определенной модели процессора. Перечислим системные регистры, поддерживаемые IA-32:
  - управляющие регистры CR0...CR4 определяют режим работы процессора и характеристики текущей исполняемой задачи;
  - регистры управления памятью GDTR, IDTR, LDTR и TR используются в защищенном режиме работы процессора для локализации управляющих структур этого режима;
  - отладочные регистры DR0...DR7 предназначены для мониторинга и управления различными аспектами отладки;
  - регистры типов областей памяти MTRR используются для аппаратного управления кэшированием в целях назначения соответствующих свойств областям памяти;
  - машинно-зависимые регистры MSR используются для управления процессором, контроля за его производительностью, получения информации об ошибках.

Почему в обозначениях многих из регистров общего назначения присутствует наклонная разделительная черта? Это не разные регистры — это части одного большого 32-разрядного регистра, но их можно использовать в программе как отдельные объекты. Зачем так сделано? Чтобы обеспечить работоспособность программ, написанных для прежних 16-разрядных моделей процессоров фирмы Intel начиная с i8086. Процессоры i486 и Pentium имеют в основном 32-разрядные регистры.

Их количество, за исключением сегментных регистров, такое же, как и у i8086, но размерность больше, что и отражено в обозначениях — они имеют приставку e (extended).

Многие из приведенных регистров предназначены для работы с определенными вычислительными подсистемами процессора: сопроцессором, MMX-расширениями. Поэтому их целесообразно рассматривать в соответствующем контексте. Так как первая часть учебника посвящена вопросам программирования целочисленной подсистемы процессора, то в данной главее описываются регистры, обеспечивающие ее функционирование. В дальнейшем при обсуждении новых вычислительных подсистем процессора будут рассмотрены и другие регистры из перечисленных ранее.

Регистры общего назначения

Регистры общего назначения используются в программах для хранения:

- операндов логических и арифметических операций;
- компонентов адреса;
- указателей на ячейки памяти.

В принципе, все эти регистры доступны для хранения операндов без особых ограничений, хотя в определенных условиях некоторые из них имеют жесткое функциональное назначение, закрепленное на уровне логики работы машинных команд. Среди всех этих регистров особо следует выделить регистр ESP. Его не следует использовать явно для хранения каких-либо операндов программы, так как в нем хранится указатель на вершину стека программы.

Все регистры этой группы позволяют обращаться к своим «младшим» частям (см. рис. 2.5). Обращение производится по их именам. Важно отметить, что использовать для самостоятельной адресации можно только младшие 16- и 8-разрядные части этих регистров. Старшие 16 битов как самостоятельные объекты недоступны. Это сделано, как мы отметили ранее, для совместимости с первыми 16-разрядными моделями процессоров фирмы Intel. Перечислим регистры, относящиеся к группе регистров общего назначения и физически находящиеся в процессоре внутри арифметико-логического устройства (поэтому их еще называют *регистрами АЛУ*):

- *регистр-аккумулятор* (Accumulator register) EAX/AX/AH/AL применяется для хранения промежуточных данных, в некоторых командах его использование обязательно;
- *базовый регистр* (Base register) EBX/BX/BH/BL применяется для хранения базового адреса некоторого объекта в памяти;
- *регистр-счетчик* (Count register) ECX/CX/CH/CL применяется в командах, производящих некоторые повторяющиеся действия. Использование регистра-счетчика зачастую скрыто в алгоритме работы той или иной команды. Например, команда организации цикла LOOP помимо передачи управления анализирует и уменьшает на единицу значение регистра ECX/CX;
- *регистр данных* (Data register) EDX/DX/DH/DL, так же как и регистр EAX/AX/AH/AL, хранит промежуточные данные (в некоторых командах его явное использование обязательно, в других он используется неявно).

Следующие два регистра предназначены для поддержки так называемых цепочечных операций, то есть операций, производящих последовательную обработку цепочек элементов, каждый из которых может иметь длину 32, 16 или 8 бит:

- *регистр индекса источника* (Source Index register) ESI/SI в цепочечных операциях содержит текущий адрес элемента в цепочке-источнике;
- *регистр индекса приемника* (Destination Index register) EDI/DI в цепочечных операциях содержит текущий адрес в цепочке-приемнике.

В архитектуре процессора на программно-аппаратном уровне поддерживается такая структура данных, как *стек*. Для работы со стеком в системе команд процессора есть специальные команды, а в программной модели процессора для этого существуют специальные регистры:

- *регистр указателя стека* (Stack Pointer register) ESP/SP содержит указатель на вершину стека в текущем сегменте стека;
- *регистр указателя базы кадра стека* (Base Pointer register) EBP/BP предназначен для организации произвольного доступа к данным внутри стека.

Не стоит пугаться столь жесткого функционального назначения регистров АЛУ. На самом деле при программировании хранить операнды команд можно в большинстве регистров, причем практически в любых сочетаниях. Однако всегда следует помнить, что некоторые команды требуют строго определенных регистров. Цель подобного жесткого закрепления регистров для этих команд — более компактная кодировка их машинного представления. Знание особенностей использования регистров машинными командами (см. приложение) позволяет, при необходимости, экономить память, занимаемую кодом программы, и более эффективно программировать алгоритм.

Сегментные регистры

Процессоры Intel аппаратно поддерживают *сегментную* организацию программы. Это означает, что любая программа состоит из трех сегментов: кода, данных и стека. Логически машинные команды в архитектуре IA-32 построены так, что при выборке каждой команды для доступа к данным программы или к стеку неявно используется информация из вполне определенных сегментных регистров. В зависимости от режима работы процессора по их содержимому определяются адреса памяти, с которых начинаются соответствующие сегменты. В программной модели IA-32 имеется шесть *сегментныхрегистров* CS, SS, DS, ES, GS, FS, служащих для доступа к четырем типам сегментов.

- *Сегмент кода* содержит команды программы. Для доступа к этому сегменту служит *регистр сегмента кода* (Code Segment register) CS. Он содержит адрес сегмента с машинными командами, к которому имеет доступ процессор (эти команды загружаются в конвейер процессора).
- *Сегмент данных* содержит обрабатываемые программой данные. Для доступа к этому сегменту *служит регистр сегмента данных* (Data Segment register) DS, который хранит адрес сегмента данных текущей программы.
- *Сегмент стека* представляет собой область памяти, называемую стеком. Работу со стеком процессор организует по следующему принципу: последний запи-

санный в эту область элемент выбирается первым. Для доступа к этой области служит *регистр сегмента стека* (Stack Segment register) SS, содержащий адрес сегмента стека.

- *Дополнительный сегмент данных.* Неявно алгоритмы выполнения большинства машинных команд предполагают, что обрабатываемые ими данные расположены в сегменте данных, адрес которого находится в регистре сегмента данных DS. Если программе недостаточно одного сегмента данных, то она имеет возможность задействовать еще три дополнительных сегмента данных. Но в отличие от основного сегмента данных, адрес которого содержится в регистре DS, при использовании дополнительных сегментов данных их адреса требуется указывать явно с помощью специальных префиксов переопределения сегментов в команде. Адреса дополнительных сегментов данных должны содержаться в *регистрах дополнительного сегмента данных* (Extension Data Segment registers) ES, GS, FS.

Регистры состояния и управления

В процессор включены два регистра (см. рис. 2.5), постоянно содержащие информацию о состоянии как самого процессора, так и программы, команды которой он в данный момент обрабатывает:

- регистр-указатель команд EIP/IP;
- регистр флагов EFLAGS/FLAGS.

С помощью этих регистров можно также ограниченным образом управлять состоянием процессора.

Регистр-указатель команд (Instruction Pointer register) EIP/IP имеет разрядность 32(16) бита и содержит смещение следующей подлежащей выполнению команды относительно содержимого регистра сегмента кода CS в текущем сегменте команд. Этот регистр непосредственно недоступен программисту, но загрузка и изменение его значения производятся различными командами управления, к которым относятся команды условных и безусловных переходов, вызова процедур и возврата из процедур. Возникновение прерываний также приводит к модификации регистра EIP/IP.

Разрядность *регистра флагов* (flag register) EFLAGS/FLAGS равна 32(16) битам. Отдельные биты данного регистра имеют определенное функциональное назначение и называются *флагами*. Младшая часть регистра EFLAGS/FLAGS полностью аналогична регистру FLAGS процессора i8086. На рис. 2.6 показано содержимое регистра EFLAGS.

Исходя из особенностей использования, флаги регистра EFLAGS/FLAGS можно разделить на три группы.

В первую группу флагов регистра EFLAGS/FLAGS входят 8 *флагов состояния*. Эти флаги могут изменяться после выполнения машинных команд. Флаги состояния регистра EFLAGS отражают особенности результата исполнения арифметических или логических операций. Это дает возможность анализировать состояние вычислительного процесса и реагировать на него с помощью команд условных переходов и вызовов подпрограмм.

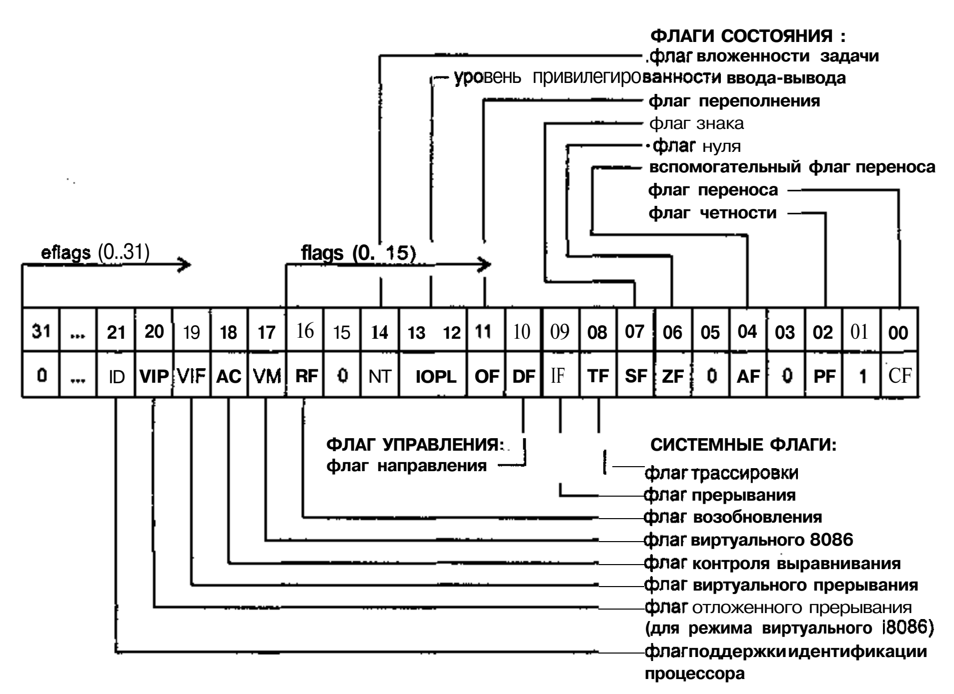


Рис. 2.6. Содержимое регистра eflags

- *Флаг переноса* (carry flag) CF:

  1 — арифметическая операция произвела перенос из старшего бита результата, старшим является 7-й, 15-й или 31-й бит в зависимости от размерности операнда;

  0 — переноса не было.

- *Флаг четности* (parity flag) PF:

  1 — 8 младших разрядов (этот флаг только для 8 младших разрядов операнда любого размера) результата содержат четное число единиц;

  0 — 8 младших разрядов результата содержат нечетное число единиц.

- *Вспомогательный флаг переноса* (auxiliary carry flag) AF применяется только для команд, работающих с BCD-числами. Фиксирует факт заема из младшей тетрады результата:

  1 — в результате операции сложения был произведен перенос из разряда 3 в старший разряд или при вычитании был заем в разряд 3 младшей тетрады из значения в старшей тетраде;

  0 — переносов и заемов в третий разряд (из третьего разряда) младшей тетрады результата не было.

- *Флаг нуля* (zero flag) ZF:

  1 — результат нулевой;

  0 — результат ненулевой.

* *Флаг знака* (sign flag) SF отражает состояние старшего бита результата (биты 7, 15 или 31 для 8-, 16- или 32-разрядных операндов соответственно):

  1 — старший бит результата равен 1;

  0 — старший бит результата равен 0.

* *Флаг переполнения* (overflow flag) OF используется для фиксации факта потери значащего бита при арифметических операциях:

  1 — в результате операции происходит перенос в старший знаковый бит результата или заем из старшего знакового бита результата (биты 7, 15 или 31 для 8-, 16- или 32-разрядных операндов соответственно);

  0 — в результате операции не происходит переноса в старший знаковый бит результата или заема из старшего знакового бита результата.

* *Уровень привилегированности ввода-вывода* (Input/Output privilege level) IOPL используется в защищенном режиме работы процессора для контроля доступа к командам ввода-вывода в зависимости от привилегированности задачи.

* *Флаг вложенности задачи* (nested task) NT используется в защищенном режиме работы процессора для фиксации того факта, что одна задача вложена в другую.

Во вторую группу флагов (группа флагов управления) регистра EFLAGS/FLAGS входит всего один *флаг направления* (directory flag) DF. Он находится в десятом бите регистра EFLAGS и используется цепочечными командами. Значение флага DF определяет направление поэлементной обработки в этих операциях: от начала строки к концу (DF = 0) либо, наоборот, от конца строки к ее началу (DF = 1). Для работы с флагом DF существуют специальные команды CLD (снять флаг DF) и STD (установить флаг DF). Применение этих команд позволяет привести флаг DF в соответствие с алгоритмом и обеспечить автоматическое увеличение или уменьшение счетчиков при выполнении операций со строками.

В третью группу флагов регистра EFLAGS/FLAGS входит 8 *системных флагов,* управляющих вводом-выводом, маскируемыми прерываниями, отладкой, переключением между задачами и режимом виртуального процессора 8086. Прикладным программам не рекомендуется модифицировать без необходимости эти флаги, так как в большинстве случаев это ведет к прерыванию работы программы. Далее перечислены системные флаги и их назначение.

* *Флаг трассировки* (trace flag) TF предназначен для организации пошаговой работы процессора:

  1 — процессор генерирует прерывание с номером 1 после выполнения каждой машинной команды (может использоваться при отладке программ, в частности отладчиками);

  0 — обычная работа.

* *Флаг прерывания* (interrupt enable flag) IF предназначен для разрешения или запрещения (маскирования) аппаратных прерываний (прерываний по входу INTR):

  1 — аппаратные прерывания разрешены;

  0 — аппаратные прерывания запрещены.

- *Флаг возобновления* (resume flag) RF используется при обработке прерываний от регистров отладки.
- *Флаг режима виртуального процессора 8086* (virtual 8086 mode) VM является признаком работы процессора в режиме виртуального 8086:

  1 — процессор работает в режиме виртуального процессора 8086;

  0 — процессор работает в реальном или защищенном режиме.
- *Флаг контроля выравнивания* (alignment check) AC предназначен для разрешения контроля выравнивания при обращениях к памяти. Используется совместно с битом AM в системном регистре CR0. К примеру, Pentium разрешает размещать команды и данные начиная с любого адреса. Если требуется контролировать выравнивание данных и команд по адресам, кратным 2 или 4, то установка данных битов приведет к тому, что все обращения по некратным адресам будут возбуждать исключительную ситуацию.
- *Флаг виртуального прерывания* (virtual interrupt flag) VIF, появившийся в процессоре Pentium, при определенных условиях (одно из которых — работа процессора в v-режиме) является аналогом флага IF. Флаг VIF используется совместно с флагом VIP.
- *Флаг отложенного виртуального прерывания* (virtual interrupt pending flag) VIP, появившийся в процессоре Pentium, устанавливается в 1 для индикации отложенного прерывания. Используется при работе в v-режиме совместно с флагом VIF.
- *Флаг идентификации* (identification flag) ID используется для того, чтобы показать факт поддержки процессором инструкции CPUID. Если программа может установить или сбросить этот флаг, это означает, что данная модель процессора поддерживает инструкцию CPUID.

Организация памяти

Физическая память, к которой процессор имеет доступ по шине адреса (см. рис. 1.2), называется *оперативной памятью* (или оперативным запоминающим устройством — ОЗУ). На самом нижнем уровне память компьютера можно рассматривать как массив *битов*. Один бит может хранить значение 0 или 1. Для физической реализации битов и работы с ними идеально подходят логические схемы. Но процессору неудобно работать с памятью на уровне битов, поэтому реально ОЗУ организовано как последовательность ячеек — *байтов*. Один байт состоит из восьми битов. Каждому байту соответствует свой уникальный адрес (его номер), называемый *физическим*. Диапазон значений физических адресов зависит от разрядности шины адреса процессора. Для i486 и Pentium он находится в пределах от 0 до $2^{32} - 1$ (4 Гбайт). Для процессоров Pentium Pro/II/III/IV этот диапазон шире — от 0 до $2^{36} - 1$ (64 Гбайт).

Механизм управления памятью полностью аппаратный. Это означает, что программа не может сама сформировать физический адрес памяти на адресной шине. Ей приходится «играть» по правилам процессора. Что это за правила, мы узнаем чуть позже. Пока же отметим, что в конечном итоге этот механизм позволяет обеспечить:

- компактность хранения адреса в машинной команде;
- гибкость механизма адресации;
- защиту адресных пространств задач в многозадачной системе;
- поддержку виртуальной памяти.

Процессор аппаратно поддерживает две модели использования оперативной памяти.

- В *сегментированной модели* программе выделяются непрерывные области памяти (сегменты), а сама программа может обращаться только к данным, которые находятся в этих сегментах.
- *Страничную модель* можно рассматривать как надстройку над сегментированной моделью. В случае использования этой модели оперативная память рассматривается как совокупность блоков фиксированного размера (4 Кбайт и более). Основное применение этой модели связано с организацией виртуальной памяти, что позволяет операционной системе использовать для работы программ пространство памяти большее, чем объем физической памяти. Для процессоров i486 и Pentium размер возможной виртуальной памяти может достигать 4 Тбайт.

Сегментированная модель памяти

Сегментация — механизм адресации, обеспечивающий существование нескольких независимых адресных пространств как в пределах одной задачи, так и в системе в целом для защиты задач от взаимного влияния. В основе механизма сегментации лежит понятие *сегмента,* который представляет собой независимый поддерживаемый на аппаратном уровне блок памяти.

Когда мы рассматривали сегментные регистры, то отмечали, что для процессоров Intel, начиная с i8086, принят особый подход к управлению памятью. Каждая программа в общем случае может состоять из любого количества сегментов, но непосредственный доступ она имеет только к трем основным сегментам (кода, данных и стека), а также к дополнительным сегментам данных числом от одного до трех. Программа никогда не знает, по каким физическим адресам будут размещены ее сегменты. Этим занимается операционная система. Операционная система размещает сегменты программы в оперативной памяти по определенным физическим адресам, после чего помещает значения этих адресов в определенные места. Куда именно, зависит от режима работы процессора. Так, в реальном режиме эти адреса помещаются непосредственно в соответствующие сегментные регистры, а в защищенном режиме они размещаются в элементы специальной системной дескрипторной таблицы. Внутри сегмента программа обращается к адресам относительно начала сегмента линейно, то есть начиная с 0 и заканчивая адресом, равным размеру сегмента. Этот относительный адрес, или *смещение,* который процессор использует для доступа к данным внутри сегмента, называется *эффективным.*

Отличия моделей сегментированной организации памяти в различных режимах хорошо видны на схеме (рис. 2.7). Различают три основных модели сегментированной организации памяти:

- сегментированная модель памяти реального режима;

- сегментированная модель памяти защищенного режима;
- сплошная модель памяти защищенного режима.

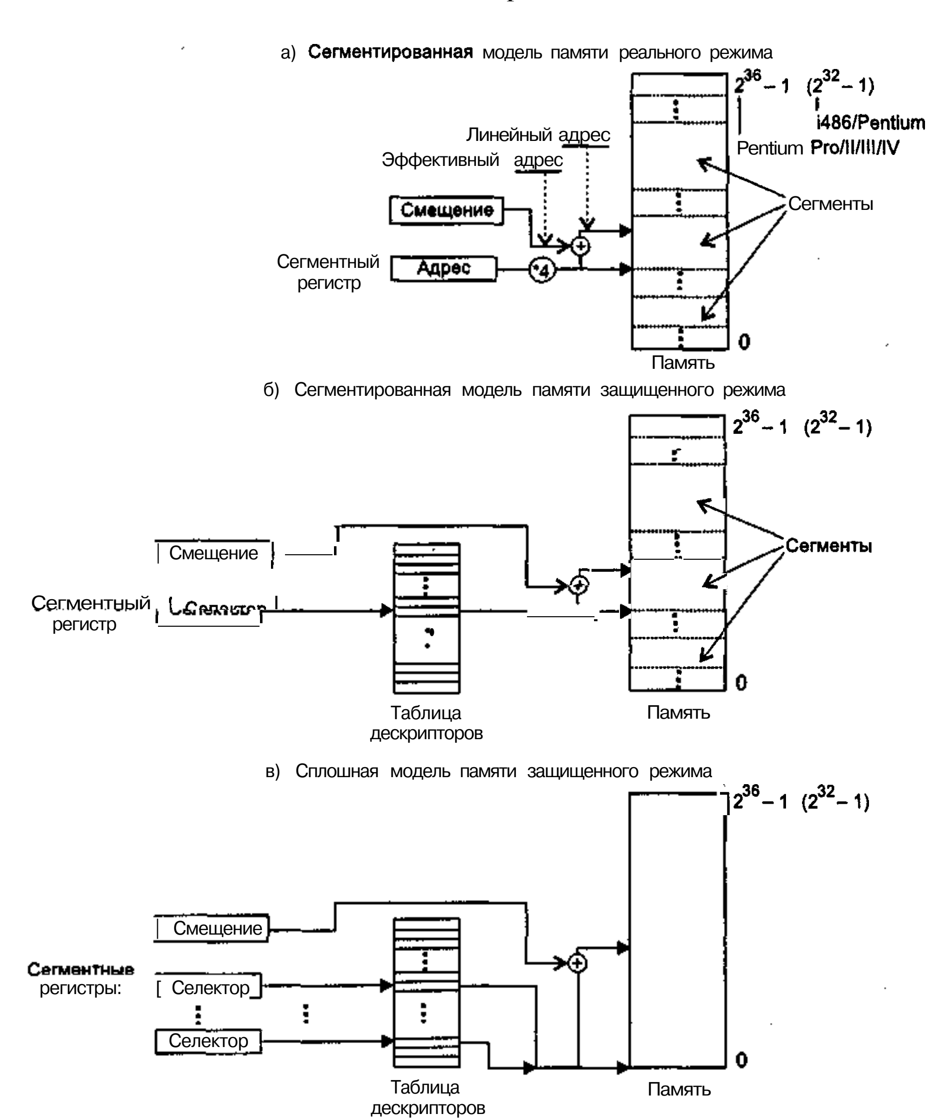


Рис. 2.7. Модели памяти процессоров Intel

Рассмотрим порядок формирования физического адреса в реальном и защищенном режимах. Уточним терминологию. Под *физическим адресом* понимается адрес памяти, выдаваемый на шину адреса процессора. Другое название этого адреса — *линейный адрес.* Подобная двойственность в названии обусловлена наличием страничной модели организации оперативной памяти. Эти названия являются синонимами только при отключении страничного преобразования адреса

(в реальном режиме страничная адресация всегда отключена). Страничная модель, как мы отметили ранее, является надстройкой над сегментированной моделью. В страничной модели линейный и физический адреса имеют разные значения. Далее мы будем обсуждать схему, на которой показан порядок формирования адреса в реальном режиме работы процессора. Обратите внимание на наличие в этой схеме устройства страничного преобразования адреса, предназначенного для того, чтобы совместить две принципиально разные модели организации оперативной памяти и выдать на шину адреса истинное значение физического адреса памяти.

Формирование физического адреса в реальном режиме

Далее перечислены характеристики механизма адресации физической памяти в реальном режиме.

- Диапазон изменения физического адреса — от 0 до 1 Мбайт. Эта величина определяется тем, что шина адреса i8086 имела 20 линий.
- Максимальный размер сегмента — 64 Кбайт. Это объясняется 16-разрядной архитектурой i8086. Нетрудно подсчитать, что максимальное значение, которое могут содержать 16-разрядные регистры, составляет 2^{16} - 1, что применительно к памяти и определяет величину 64 Кбайт.
- Для обращения к конкретному физическому адресу оперативной памяти необходимо определить адрес начала сегмента (сегментную составляющую) и смещение внутри сегмента.

Понятие *адреса начала сегмента* ввиду принципиальной важности требует дополнительного пояснения. Исходя из разрядности сегментных регистров, можно утверждать, что сегментная составляющая адреса (или *база сегмента)* представляет собой всего лишь 16-разрядное значение, помещенное в один из сегментных регистров. Максимальное значение, которое при этом получается, соответствует 2^{16} - 1. Если так рассуждать, то получается, что адрес начала сегмента может быть только в диапазоне 0-64 Кбайт от начала оперативной памяти. Возникает вопрос, как адресовать остальную часть оперативной памяти вплоть до 1 Мбайт с учетом того, что размер самого сегмента не превышает 64 Кбайт. Дело в том, что в сегментном регистре содержатся только старшие 16 битов физического адреса начала сегмента. Недостающие младшие четыре бита 20-разрядного адреса получаются сдвигом значения в сегментном регистре влево на 4 разряда. Эта операция сдвига выполняется аппаратно и для программного обеспечения абсолютно прозрачна. Получившееся 20-разрядное значение и является настоящим физическим адресом, соответствующим началу сегмента. Что касается второго компонента (*смещения),* участвующего в образовании физического адреса некоторого объекта в памяти, то он представляет собой 16-разрядное значение. Это значение может содержаться явно в команде либо косвенно в одном из регистров общего назначения. В процессоре эти две составляющие складываются на аппаратном уровне, в результате получается физический адрес памяти размерностью 20 битов. Данный механизм образования физического адреса позволяет сделать программное обеспечение перемещаемым, то есть не зависящим от конкретных адресов загрузки его в оперативной памяти (рис. 2.8).

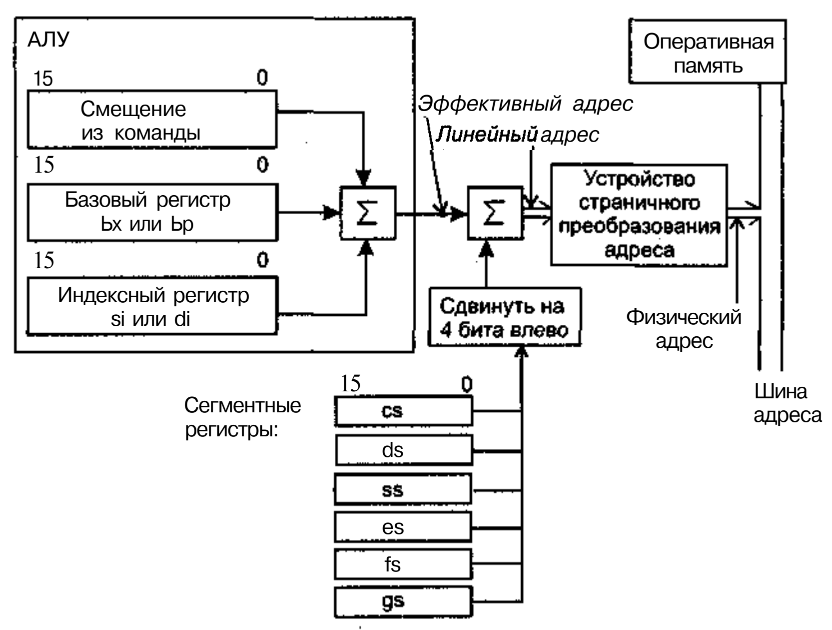


Рис. 2.8. Механизм формирования физического адреса в реальном режиме

На рисунке хорошо видно, как формируется некоторый целевой физический адрес: сегментная часть извлекается из одного из сегментных регистров, сдвигается на четыре разряда влево и суммируется со смещением. В свою очередь, видно, что значение смещения можно получить минимум из одного и максимум из трех источников: из значения смещения в самой машинной команде и/или из содержимого одного базового и/или одного индексного регистра. Количество источников, участвующих в формировании смещения, определяется кодированием конкретной машинной команды, и если таких источников несколько, то значения в них складываются. В заключение заметим, что не стоит волноваться из-за несоответствия размеров шины адреса процессора i486 или Pentium (32 бита) и 20-разрядного значения физического адреса реального режима. Пока процессор находится в реальном режиме, старшие 12 линий шины адреса попросту недоступны, хотя при определенных условиях и существует возможность работы с первыми 64 Кбайт оперативной памяти, лежащими сразу после первого мегабайта.

Недостатки такой организации памяти:

- сегменты бесконтрольно размещаются с любого адреса, кратного 16 (так как содержимое сегментного регистра аппаратно смещается на 4 разряда), и, как следствие, программа может обращаться по любым адресам, в том числе и реально не существующим;
- сегменты имеют максимальный размер 64 Кбайт;
- сегменты могут перекрываться другими сегментами.

Желанием ввести в архитектуру средства, позволяющие избавиться от указанных недостатков, и обусловлено, в частности, появление защищенного режима,

в котором работают все современные операционные системы, в том числе Windows и Linux.

Формирование физического адреса в защищенном режиме

Основная идея защищенного режима — защитить исполняемые процессором программы от взаимного влияния. В защищенном режиме процессор поддерживает два типа защиты — по привилегиям и по доступу к памяти. В контексте нашего изложения интерес представляет второй тип защиты. Его мы и рассмотрим.

Для введения любого механизма защиты нужно иметь как можно больше информации об объектах защиты. Для процессора такими объектами являются исполняемые им программы. Организуя защиту программ по доступу к памяти, фирма Intel не стала нарушать принцип сегментации, свойственный ее процессорам. Так как каждая программа занимает один или несколько сегментов в памяти, то логично иметь больше информации об этих сегментах как об объектах, реально существующих в данный момент в вычислительной системе. Если каждому из сегментов присвоить определенные атрибуты, то часть функций по контролю за доступом к ним можно переложить на процессор. Что и было сделано. Любой сегмент памяти в защищенном режиме имеет следующие атрибуты:

- расположение сегмента в памяти;
- размер сегмента;
- уровень привилегий (определяет права данного сегмента относительно других сегментов);
- тип доступа (определяет назначение сегмента);
- некоторые другие.

В отличие от реального режима, в защищенном режиме программа уже не может запросто обратиться по любому физическому адресу памяти. Для этого она должна иметь определенные полномочия и удовлетворять ряду требований.

Ключевым объектом защищенного режима является специальная структура — *дескриптор сегмента,* который представляет собой 8-байтовый дескриптор (крат-

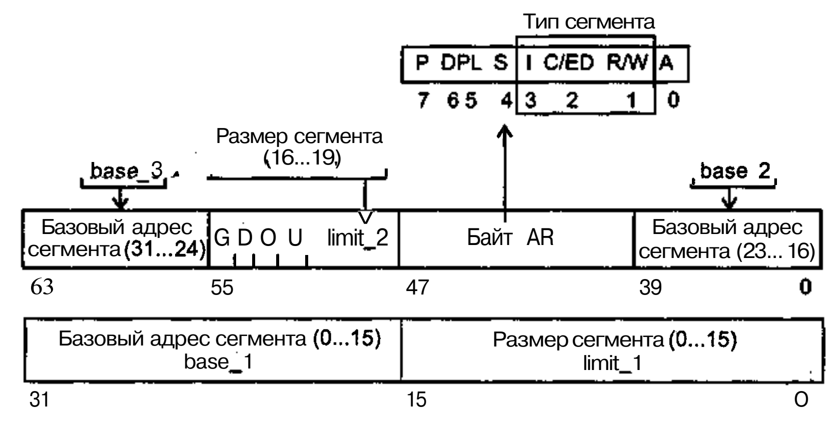


Рис. 2.9. Структура дескриптора сегмента защищенного режима процессора

кое описание) непрерывной области памяти, содержащий перечисленные ранее атрибуты. На рис. 2.9 приведена структура дескриптор сегмента.

Приведем назначение некоторых полей дескриптора сегмента:

- limit_1 и limit_2 — 20-разрядное поле, определяющее размер сегмента;
- base_1 и base_2 — 32-разрядное поле, определяющее значение линейного адреса начала сегмента в памяти;
- AR — байт, поля которого определяют права доступа к сегменту;
- D — бит разрядности операндов и адресов (глава 3);
- G — бит гранулярности.

В защищенном режиме размер сегмента не фиксирован, его расположение можно задать в пределах 4 Гбайт. Если посмотреть на рисунок, то возникнет вопрос: почему разорваны поля, определяющие размер сегмента и его начальный (базовый) адрес? Это результат эволюции процессоров. Защищенный режим впервые появился в процессоре i80286. Этот процессор имел 24-разрядную адресную шину и, соответственно, мог адресовать в защищенном режиме до 16 Мбайт оперативной памяти. Для этого ему достаточно было иметь в дескрипторе поле базового адреса 24 бита и поле размера сегмента 16 битов. После появления процессора i80386 с 32-разрядной шиной команд и данных в целях совместимости программ разработчики не стали менять формат дескриптора, а просто использовали свободные поля. Внутри процессора эти поля объединены. Внешне же они остались разделенными, и при программировании с этим приходится мириться.

Следующий интересный момент связан с тем, что размер сегмента в защищенном режиме может достигать 4 Гбайт, то есть занимать все доступное физическое пространство памяти. Как это возможно, если суммарный размер поля размера сегмента составляет всего 20 битов, что соответствует величине 1 Мбайт? Секрет скрыт в поле *гранулярности* — бит G (см. рис. 2.9). Если G = 0, то значение в поле размера сегмента означает размер сегмента в байтах, если G - 1, — то в страницах. Размер страницы составляет 4 Кбайт. Нетрудно подсчитать, что когда максимальное значение поля размера сегмента составляет 0fffffh,то это соответствует 1 М страниц или величине 1 М х 4 Кбайт = 4 Гбайт.

Выведение информации о базовом адресе сегмента и его размере на уровень процессора позволяет аппаратно контролировать работу программ с памятью и предотвращать обращения к несуществующим адресам либо к адресам, находящимся вне предела, разрешенного полем размера сегмента limit.

Другой аспект защиты заключается в том, что сегменты неравноправны в правах доступа к ним. Информация об этом содержится в специальном байте AR, входящем в состав дескриптора. Наиболее важные поля байта AR — это dpl и биты R/W, C/ED и I, которые вместе определяют тип сегмента. Поле dpl — часть механизма защиты по привилегиям. Суть этого механизма заключается в том, что конкретный сегмент может находиться на одном из четырех уровней привилегированности с номерами О, 1, 2 и 3. Самым привилегированным является уровень 0. Существует ряд ограничений (опять-таки на аппаратном уровне) на взаимодействие сегментов кода, данных и стека с различными уровнями привилегий.

Итак, мы выяснили, что в защищенном режиме перед использованием любой области памяти должна быть проведена определенная работа по инициализации соответствующего дескриптора. Эту работу выполняет операционная система или программа, сегменты которой также описываются подобными дескрипторами. Все дескрипторы собираются вместе в одну из трех дескрипторных таблиц:

- ***глобальная дескрипторная*** *таблица* (Global Descriptor Table, GDT), ее адрес хранится в регистре GDTR;
- ***локальная*** *дескрипторная таблица* (Local Descriptor Table, LDT), ее адрес хранится в регистре LDTR;
- *дескрипторная таблица векторов прерываний* (Interrupt Descriptor Table, IDT), ее адрес хранится в регистре IDTR.

В какую именно таблицу должен быть помещен дескриптор, определяется его назначением. Адрес, по которому размещаются эти дескрипторные таблицы, может быть любым; он хранится в специально предназначенном для этого адреса системном регистре.

Схемы, показанные на рис. 2.7, *б* и *в,* иллюстрируют Принцип формирования адреса в защищенном режиме. Важно отметить изменение роли сегментных регистров. В защищенном режиме они содержат не адрес, а *селектор,* то есть указатель на соответствующую ячейку одной из дескрипторных таблиц (GDT или LDT).

Остальные элементы архитектуры IA-32, такие как формат машинных команд, типы данных и др., логично рассмотреть в следующих главах в контексте соответствующих аспектов использования языка ассемблера.

Итоги

- Понимание архитектуры ЭВМ является ключевым для изучения ассемблера. Это касается любого типа компьютера. Структура ассемблера, формат его команд, адресация операндов и т. д. полностью отражают особенности архитектуры компьютера. Есть общие архитектурные свойства, присущие всем современным машинам фон-неймановской архитектуры, и есть частные свойства, присущие конкретному типу компьютеров.
- Целью изучения архитектуры является:
  - выявление набора доступных для программирования регистров, их функционального назначения и структуры;
  - понимание организации оперативной памяти и порядка ее использования;
  - знакомство с типами данных;
  - изучение формата машинных команд;
  - выяснение организации обработки прерываний.
- Процессор содержит 32 доступных тем или иным образом регистра. Они делятся на пользовательские и системные.
- Пользовательские регистры имеют определенное функциональное назначение. Среди них особо нужно выделить регистр флагов EFLAGS и регистр указателя команды EIP. Назначение регистра EFLAGS — отражать состояние процессора

после выполнения последней машинной команды. Регистр EIP содержит адрес следующей выполняемой машинной команды. Доступ к этим регистрам, в силу их специфики, со стороны программ пользователя ограничен.

- Процессор имеет три основных режима работы:
  - реальный режим, который использовался для i8086 и поддерживается до сих пор для обеспечения совместимости программного обеспечения;
  - защищенный режим, который впервые появился в i80286;
  - режим виртуального процессора i8086 обеспечивает полную эмуляцию i8086, позволяя при этом организовать многозадачную работу нескольких таких программ.
- Процессор имеет сложную систему управления памятью, функционирование которой зависит от режима работы процессора.

Глава 3

Система команд процессора IA-32

- Формат машинных команд IA-32
- Назначение и интерпретация полей машинной команды
- Основы декодирования машинных команд
- Функциональная классификация машинных команд

Система машинных команд является важнейшей частью архитектуры компьютера, так как с их помощью производится непосредственное управление работой процессора. К примеру, система команд процессора Pentium IV содержит более 300 машинных команд. С появлением каждой новой модели процессора количество команд, как правило, возрастает, отражая архитектурные новшества данной модели по сравнению с предшествующими.

При знакомстве с системой машинных команд необходимо учитывать два аспекта — собственно *набор* машинных команд и правила представления этих команд на уровне процессора, то есть *формат* машинных команд. Процессору компьютера понятен только один язык — язык машинных команд. Машинные команды представляют собой сформированные по определенным правилам последовательности нулей и единиц. Для того чтобы заставить процессор выполнить некоторое действие, ему нужно выдать соответствующее указание в виде машинной команды, а для выполнения более сложной работы достаточно написать программу в двоичных кодах. Программирование первых компьютеров осуществлялось именно таким способом. Недостатки процесса написания программ в двоичном коде очевидны. Для облегчения процесса разработки программ был придуман язык ассемблера, как символический аналог машинного языка, а в архитектуру компьютера введен блок микропрограммного управления. Для каждой машинной команды блок

микропрограммного управления содержит отдельную микропрограмму, с помощью которой действия, заданные этой командой, переводятся на язык сигналов, направляемых нужным подсистемам процессора. После этих нововведений процесс разработки программы значительно упростился. Человек пишет программу на понятном ему языке символов, специальная программа — ассемблер — переводит (транслирует) программу человека на машинный язык, а блок микропрограммного управления нужным образом интерпретирует машинные команды для процессора, процессор выполняет нужную работу.

В дальнейшем, с появлением программного обеспечения более высокого уровня, язык ассемблера не потерял своей роли, а наоборот, приобрел новые качества. В силу иерархичности программного обеспечения компьютера ассемблер стал промежуточным, связующим звеном между разнородным и разноязыким программным обеспечением более высокого уровня и процессором.

Таким образом, существует взаимно однозначное соответствие машинных команд и команд ассемблера. Понимание правил формирования машинных команд из команд ассемблера является одним из необходимых условий не только для изучения языка ассемблера, но и для понимания логики работы компьютера в целом.

Формат машинных команд IA-32

Машинная команда представляет собой закодированное по определенным правилам указание процессору на выполнение некоторой операции. Правила кодирования команд называются *форматом команд*. Команды процессоров архитектуры IA-32 считаются сложными. Максимальная длина машинной команды IA-32 составляет 15 байт. Реальная команда может содержать гораздо меньшее количество полей, вплоть до одного — только код операции. Приведенный на рис. 3.1 формат машинной команды является наиболее полным.

Как на уровне формата машинной команды соответствуют между собой машинные команды и команды ассемблера? Для примера рассмотрим типичную команду языка ассемблера:

```
mov ebx, eax
```

Команда MOV производит копирование содержимого регистра EAX в регистр EBX. Соответствующая машинная команда будет выглядеть так:

```
8B D8
```

Значение 8B — код операции. Еще один пример команды MOV:

```
mov ecx, 128
```

Данная команда инициализирует содержимое регистра ECX десятичным значением 128. Соответствующая машинная команда выглядит так:

```
B9 00000080
```

Значение поля с кодом операции — B9. Из примеров видно, что прямого соответствия между структурой команды ассемблера и соответствующей машинной командой нет. Несмотря на то, что команда ассемблера одна и та же (MOV), коды машинных команд — разные (8B и B9). Большинство команд ассемблера имеют несколько возможных вариантов сочетания операндов. Как показано в приведенных примерах, несмотря на одинаковые названия команд ассемблера, для

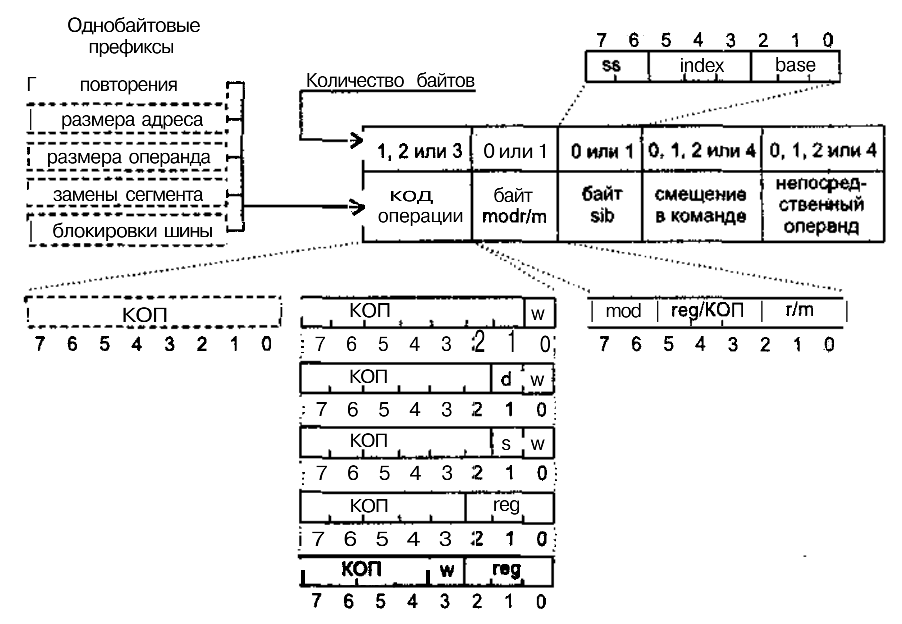


Рис. 3.1. Формат машинной команды

каждого возможного сочетания операндов имеется своя машинная команда, со своим значением поля кода операции (рис. 3.1). Это говорит о том, что машинная команда всегда однозначна по отношению к производимым ею действиям на уровне аппаратуры. Несколько упрощая реальность, можно утверждать, что значение в поле кода операции является номером микропрограммы в блоке микропрограммного управления для каждой конкретной команды ассемблера с каждым конкретным вариантом сочетания операндов.

Логически любая команда языка ассемблера содержит несколько элементов.

- Элемент, описывающий, *что делать,* называется *кодом операции* (КОП). Значение в поле кода операции некоторым образом определяет в блоке микропрограммного управления подпрограмму, реализующую действия для данной команды.
- Элементы, описывающие *объекты, с которыми нужно что-то делать*, являются *операндами.* Операнды в команде могут и не задаваться, а подразумеваться по умолчанию.
- Элементы, описывающие, *как делать*, являются *типами операндов* и обычно задаются неявно. Они используются транслятором ассемблера при формировании машинной команды для определения значения поля кода операции.

Эти же элементы имеет и машинная команда, но в закодированном виде. Перевод команд ассемблера в соответствующие машинные команды осуществляет специальная программа — *ассемблер,* которую можно также назвать транслятором (компилятором) ассемблера.

Опишем назначения полей машинной команды (рис. 3.1).

Поле префиксов

Префиксы — необязательные однобайтные элементы машинной команды. Назначение префиксов — изменить действия, выполняемые командой. Префиксы могут указываться программистом явно при написании исходного текста программы, либо их, по определенным соображениям, может вставить ассемблер. Процессор распознает префиксы по их значениям. Машинная команда может иметь до четырех префиксов одновременно. В памяти префиксы предшествуют команде. Порядок их следования при этом может быть любым.

Далее перечислены типы префиксов, которые может использовать прикладная программа.

- *Префикс замены сегмента* в явной форме указывает, какой сегментный регистр используется в данной команде для адресации стека или данных. Префикс отменяет выбор сегментного регистра по умолчанию. Префиксы замены сегмента имеют следующие значения:
  - 2Eh — замена сегмента CS;
  - 36h — замена сегмента SS;
  - 3Eh — замена сегмента DS;
  - 26h — замена сегмента ES;
  - 64h — замена сегмента FS;
  - 65h — замена сегмента GS.
- *Префикс повторения* используется с цепочечными командами (командами обработки строк). Этот префикс «зацикливает» команду для обработки всех элементов цепочки. Система команд поддерживает два типа префиксов: *безусловные* (REP — 0F3h), заставляющие цепочечную команду повторяться некоторое количество раз, и *условные* (REPE/REPZ — 0F3h, REPNE/REPNZ — 0F2h), которые при зацикливании проверяют некоторые флаги, и в результате проверки возможен досрочный выход из цикла.
- *Префикс блокировки шины* инициирует выдачу процессором сигнала LOCK# (значение 0F0h) для блокировки системной шины. Используется в многопроцессорных конфигурациях для обеспечения монопольного владения системной шиной. Сигнал LOCK# может формироваться лишь с определенной номенклатурой команд процессора, работающих в цикле «чтение-модификация-запись».
- *Префикс размера адреса* (значение 67h) уточняет разрядность адреса: 16 или 32 бита. Каждой команде, в которой используется адресный операнд, ставится в соответствие разрядность адреса этого операнда. Если разрядность адреса для данной команды составляет 16 битов, это означает, что команда содержит 16-разрядное смещение и оно соответствует 16-разрядному смещению адресного операнда относительно начала некоторого сегмента (см. рис. 3.1). В контексте материала главы 2 (см. рис. 2.7 и 2.8) это смещение называется *эффективным адресом*. Если разрядность адреса составляет 32 бита, это означает, что команда содержит 32-разрядное смещение, оно соответствует 32-разрядному смещению адресного операнда относительно начала сегмента и по его значению формиру-

ется 32-разрядное смещение в сегменте (см. рис. 3.1). С помощью префикса разрядности адреса можно изменить действующее по умолчанию значение разрядности адреса. Это изменение будет касаться только той команды, которой предшествует префикс.

- *Префикс размера операнда* (значение 66h) аналогичен префиксу размера адреса, но указывает на разрядность операндов (32 или 16 битов), с которыми работает команда.

По каким правилам устанавливаются по умолчанию значения атрибутов разрядности адреса и операндов? Если команда имеет операнд в памяти, то его адрес представляет собой значение смещения относительно начала сегмента данных (если не используется префикс переопределения сегмента) и содержится в поле смещения машинной команды. Размер этого поля зависит от текущего режима адресации (атрибуты use16 или use32 в директивах сегментации). При 16-разрядной адресации размер поля смещения в машинной команде составляет 16 битов. При 32-разрядной адресации размер поля смещения в машинной команде составляет 32 бита. Явное задание префикса размера адреса позволяет указать процессору значение, отличающееся от действующего по умолчанию. Например, если действующий размер адреса равен 16 битам, то использование перед какой-либо командой префикса 67h определит для нее (и только для нее!) размер адреса в 32 бита. И наоборот, если действующий размер адреса равен 32 бита, то указание перед командой префикса 67h определит для нее (и только для нее!) размер адреса в 16 битов. Физически это будет отражаться на размере поля смещения в данной машинной команде.

Префикс размера операнда 66h позволяет сменить действующий для данной команды атрибут размера операнда. Команда, которая по умолчанию работает со словом (16 битов) или с двойным словом (32 бита), имеет атрибут размера операнда, равный 16 и 32 бита соответственно. Применение префикса размера операнда позволяет сменить действующий по умолчанию атрибут.

При работе процессора i8086 в реальном и виртуальном режимах атрибуты размера адреса и операнда по умолчанию равны 16 битов. В защищенном режиме значения этих атрибутов зависят от значения бита D дескриптора сегмента. Если D = 0, то атрибуты размера адреса и операнда равны 16 битов, если D = 1, то эти атрибуты равны 32 бита. Изменить действующие по умолчанию атрибуты адреса и размера операндов можно применением атрибутов use16 или use32 в директивах сегментации.

В реальном режиме с помощью префикса разрядности адреса можно задействовать 32-разрядную адресацию, но при этом необходимо помнить об ограниченности размера сегмента величиной 64 Кбайт. Аналогично префиксу разрядности адреса можно использовать префикс разрядности операнда в реальном режиме для работы с 32-разрядными операндами (к примеру, в арифметических командах).

В команде префиксы размера адреса и операнда могут указываться одновременно. Каким образом комбинация префиксов (указанных явно и установленных по умолчанию) влияет на атрибуты размера операнда и адреса машинной команды, показано в табл. 3.1. В ней отражено и то, как влияет на эти атрибуты состояние бита D дескриптора сегмента в защищенном режиме. Строки таблицы, соот-

ветствующие нулевому значению бита D, используются в реальном и виртуальном режимах работы процессора i8086.

Необходимо обратить внимание на то, что команды работы со стеком также имеют аналогичные атрибуты размера и адреса операнда. Атрибут размера адреса влияет на выбор регистра — указателя стека: при размере адреса 16 битов используется регистр SP, при размере адреса 32 бита используется регистр ESP. Аналогично влияет на работу команд со стеком префикс размера операнда: при использовании префикса размера операнда 16 битов операнд в стеке трактуется как 16-разрядный, при использовании префикса размера операнда 32 бита операнд в стеке трактуется как 32-разрядный.

Таблица 3.1. Значения атрибутов размеров операнда и адреса

Бит D	66h	67h	Размер операнда	Размер адреса
0	–	–	16	16
0	–	+	16	32
0	+	–	32	16
0	+	+	32	32
1	–	–	32	32
1	–	+	32	16
1	+	–	16	32
1	+	+	16	16

В качестве примера префикса, который при формировании машинной команды вставляет сам ассемблер, можно привести префикс со значением 0FFh. Он называется *префиксом смены алфавита* и извещает о том, что поле кода операции в данной команде двухбайтовое.

Код операции

Код операции — обязательный элемент, описывающий операцию, выполняемую командой. Код операции может занимать от одного до трех байт. Для некоторых машинных команд часть битов кода операции может находиться в байте mod r/m.

Многим командам соответствует несколько кодов операций, каждый из которых определяет нюансы выполнения операции. Отметим, что поле кода операции не имеет однозначной структуры (см. рис. 3.1). В зависимости от конкретных команд, не обязательно разных с точки зрения языка ассемблера, оно может иметь в своем составе от одного до трех элементов, назначение которых описано в табл. 3.2. Один из этих трех элементов является непосредственно кодом операции или ее частью, остальные уточняют детали операции. Такое строение поля кода операции усложняет, в частности, процесс дизассемблирования. Для определения размера и границ очередной команды необходимо полностью проанализировать ее поле кода операции.

Таблица 3.2. Назначение дополнительных битов поля кода операции

Поле	Количество	Назначение
d	1	Определяет направление передачи данных: 0 — передача данных из регистра reg в память (или регистр), адресуемую полем *m/m;* 1 — передача данных из памяти (или регистра), адресуемой нолем r/m, в регистр reg. При наличии байта sib адрес операнда в памяти формируется с учетом содержимого этого байта
s	1	Задает необходимость расширения 8-разрядного непосредственного операнда до 16 или 32 бита. Старшие биты при этом заполняются значением старшего (знакового) бита исходного 8-разрядного операнда
w	1	Определяет размер данных, которыми оперирует команда: байт, слово, двойное слово: 0 — 8 битов; 1 — 16 битов для 16-разрядного размера операндов или 32 бита для 32-разрядного размера операндов
reg	3	Определяет регистр, используемый в команде. Значение поля зависит от ноля w, в том числе если поле w отсутствует (см. следующий подраздел)

Последующие поля машинной команды определяют характеристики и местоположение операндов, участвующих в операции, и особенности их использования (см. далее).

Байт режима адресации mod r/m

Байт режима адресации mod r/m, иногда называемый *постбайтом,* несет информацию об операндах и режиме адресации. Большинство команд процессора Intel — двухоперандные. Операнды могут находиться в памяти, а также в одном или двух регистрах. Архитектура IA-32 не допускает, чтобы оба операнда команды находились в памяти. Если операнд находится в памяти, то байт mod r/m определяет компоненты (смещение, базовый и индексный регистры), используемые для вычисления его эффективного адреса (см. главу 2). Байт mod r/m состоит из трех полей (см. рис. 3.1).

- Поле mod (два бита) определяет способ адресации и количество байтов, занимаемых в команде адресом операнда (поле смещения в команде). Поле mod используется совместно с полем r/m, которое определяет способ модификации адреса операнда полем смещения в команде. Поле mod в комбинации с полем r/m образует 32 возможных значения, обозначающих один из восьми регистров и 24 режима адресации. К примеру, если mod = 00, то поле смещения в команде отсутствует и адрес операнда определяется содержимым базового и/или индексного регистра. Какие именно регистры потребуются для вычисления эффективного адреса, определяется значением этого байта. Если mod = 01, то поле смещения в команде присутствует, занимает один байт и модифицируется содержимым базового и/или индексного регистра. Если mod = 10, то поле смещения в команде присутствует, занимает два или четыре байта (в зависимости от значения, действующего по умолчанию или определяемого префиксом размера адреса)

и модифицируется содержимым базового и/или индексного регистра. Если mod = 11, то операндов в памяти нет — они находятся в регистрах. Это же значение байта mod используется в случае, когда команда работает с непосредственным операндом.

* Поле reg (3 бита) определяет либо регистр (табл. 3.3 и 3.4), находящийся в команде на месте *второго* операнда, либо возможное расширение кода операции (давая в совокупности размер поля КОП в 11 битов).

я Поле r/m используется совместно с полем mod и определяет либо регистр, находящийся в команде на месте *первого* операнда (если mod = 11), либо базовые и индексные регистры, применяемые для вычисления эффективного адреса (совместно с полем смещения в команде).

Таблица 3.3. Значения кодов в поле reg (поле w присутствует в команде)

Поле reg	w = 0	w = 1
000	AL	AX/EAX
001	CL	CX/ECX
010	DL	DX/EDX
011	BL	BX/EBX
100	AH	SP/ESP
101	CH	BP/EBP
no	DH	SI/ESI
111	BH	DI/EDI

Таблица 3.4. Значения кодов в поле reg (поле w отсутствует)

Поле reg	**16-разрядные** операции	32-разрядные операции
000	AX	EAX
001	CX	ECX
010	DX	EDX
OH	BX	EBX
100	SP	ESP
101	BP	EBP
HO	SI	ESI
111	DI	EDI

В таблицах нет содержимого поля reg для 16-разрядных регистров в 32-разрядных операциях, так как в архитектуре Intel отдельно использовать старшую половину 32-разрядного регистра невозможно.

В архитектуре Intel один из операндов обязательно находится в регистре, и он может быть *первым* или *вторым.* Расположение *первого* и *второго* операндов в формате команды фиксировано. Но, например, команда MOV может выполнять пересылку как из регистра в память, так и из памяти в регистр. В машинном представлении

3 Зак. 256

это одна и та же команда. В ее поле reg будет содержаться код регистра (см. табл. 3.3 и 3.4), а в поле r/m — код режима адресации (см. далее). Эти две команды будут различаться только одним битом d, который определяет направление передачи. Если в команде участвуют два регистра, то в этом случае вступает в силу правило: поле reg определяет *второй* операнд, а поле r/m — *первый*. Если команда mov работает с ячейкой памяти, то в исходном тексте программы могут быть следующие варианты записи этой команды:

```
mov  abl1,ax    ;пересылка содержимого ах в ячейку памяти abll
```

или

```
mov  ax,abl1    ;пересылка содержимого ячейки памяти abll в ах
```

В машинном представлении эти две команды будут выглядеть одинаково, за исключением бита d:

- для команды MOV abll,ax бит d = 0;
- для команды MOV ax,abll бит d = 1.

Наиболее сложными для декодирования являются команды с операндом в памяти. Фирма Intel сопровождает описание системы команд специальными таблицами, облегчающими интерпретацию содержимого байта mod r/m (табл. 3.5 и 3.6). С их помощью можно довольно легко восстановить компоненты, из которых формировался адрес операнда, и, в конечном итоге, восстановить соответствующую команду ассемблера для данной машинной команды.

Таблица 3.5. Значения байта mod r/m (16-разрядная адресация)

r8			AL	CL	DL	BL	AH	CH	DH	BH
r16			AX	CX	DX	BX	SP	BP	SI	DI
r32			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
/цифра (код операции)			0	1	2	3	4	5	6	7
reg			000	001	010	on	100	101	110	111
Эффективный адрес	mod	r/m	**Шестнадцатеричные** значения mod r/m							
[BX+SI]	00	000	00	08	10	18	20	28	30	38
[BX+DI]	00	001	01	09	11	19	21	29	31	39
[BP+SI]	00	010	02	Oa	12	la	22	2a	32	3a
[BP+DI]	00	011	03	Ob	13	1b	23	2b	33	3b
[SI]	00	100	04	Oc	14	1c	24	2c	34	3c
[DI]	00	101	05	Od	15	1d	25	2d	35	3d
смещ_16	00	ПО	06	Oe	16	1e	26	2e	36	3e
[BX]	00	111	07	Of	17	1f	27	2f	37	3f
[BX+SI]+смещ_8	01	000	40	48	50	58	60	68	70	78
[BX+DI]+смещ_8	01	001	41	49	51	59	61	69	71	79
[BP+SI]+смещ_8	01	010	42	4a	52	5a	62	6a	72	7a
[BP+DI]+смещ_8	01	011	43	4b	53	5b	63	6b	73	7b
[SI]+смещ_8	01	100	44	4c	54	5c	64	6c	74	7c

Эффективный адрес	mod	r/m	Шестнадцатеричные значения mod г/т							
[DI]+смещ_8	01	101	45	4d	55	5d	65	6d	75	7d
[BP]+смещ_8	01	110	46	4e	56	5e	66	6e	76	7e
[BX]+смещ_8	01	111	47	4f	57	5f	67	6f	77	7f
[BX+SI]+смещ_16	10	000	80	88	90	98	a0	a8	b0	b8
[BX+DI]+смещ_16	10	001	81	89	91	99	a1	a9	b1	b9
[BP+SI]+смещ_16	10	010	82	8a	92	9a	a2	aa	b2	ba
[BP+DI]+смещ_16	10	011	83	8b	93	9b	a3	ab	b3	bb
[SI]+смещ_16	10	100	84	8c	94	9c	a4	ac	b4	bc
[DI]+смещ_16	10	101	85	8d	95	9d	a5	ad	b5	bd
[BP]+смещ_16	10	110	86	8e	96	9e	a6	ae	b6	be
[BX]+смещ_16	10	111	87	8f	97	9f	a7	af	77	bf
EAX/AX/AL	11	000	c0	c8	d0	d8	e0	e8	ГО	f8
ECX/CX/CL	11	001	c1	c9	d1	d9	e1	e9	f1	f9
EDX/DX/DL	11	010	c2	ca	d2	Da	e2	ea	f2	fa
EBX/BX/BL	И	011	c3	cb	d3	Db	e3	eb	G	fb
ESP/SP/AH	11	100	c4	cc	d4	DC	e4	ec	f4	fc
EBP/BP/CH	11	101	c5	cd	d5	Dd	e5	ed	fS	fd
ESI/SI/DH	11	ПО	c6	ce	d6	De	e6	ee	f6	fe
EDI/DI/BH	11	111	c7	cf	d7	Df	e7	ef	f7	ff

Таблица 3.6. Значения байта mod r/m (32-разрядная адресация)

r32			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
/цифра (код операции)			0	1	2	3	4	5	6	7
reg			000	001	010	011	100	101	ПО	111
Эффективный адрес	mod	r/m	Шестнадцатеричные значения mod r/m							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]	00	001	01	09	11	19	21	29	31	39
[EDX]	00	010	02	0a	12	1a	22	2a	32	3a
[EBX]	00	011	03	0b	13	1b	23	2b	33	3b
[sib]	00	100	04	0c	14	1c	24	2c	34	3c
Смещ_32	00	101	05	0d	15	1d	25	2d	35	3d
[ESI]	00	ПО	06	0e	16	1e	26	2e	36	3e
[EDI]	00	111	07	0f	17	1f	27	2f	37	3f
[EAX]+смещ_8	01	000	40	48	50	58	60	68	70	78
[ECX]+смещ_8	01	001	41	49	51	59	61	69	71	79
[EDX]+смещ_8	01	010	42	4a	52	5a	62	6a	72	7a

продолжение ➩

Таблица 3.6 *(продолжение)*

Эффективный адрес	mod	r/m	Шестнадцатеричные значения mod r/m							
[EBX]+смещ_8	01	011	43	4b	53	5b	63	6b	73	7b
[sib]+смещ 8	01	100	44	4c	54	5c	64	6c	74	7c
[EBP]+смещ_8	01	101	45	4d	55	5d	65	6d	75	7d
[ESI]+смещ_8	01	ПО	46	4e	56	5e	66	6e	76	7e
[EDI]+смещ_8	01	111	47	4f .	57	5f	67	6f	77	7f
[EAX]+смещ_32	10	000	80	88	90	98	a0	a8	b0	b8
[ECX]+смещ_32	10	001	81	89	91	99	a1	a9	b1	b9
[EDX]+смещ_32	10	010	82	8a	92	9a	a2	aa	b2	ba
[EBX]+смещ_32	10	011	83	8b	93	9b	a3	ab	b3	bb
[SIB]	10	100	84	8c	94	9c	a4	ac	b4	bc
[EBP]+смещ_32	10	101	85	8d	95	9d	a5	ad	b5	bd
[ESI]+смещ_32	10	ПО	86	8e	96	9e	a6	ae	b6	be
[EDI]+смещ_32	10	111	87	8f	97	9f	a7	af	77	bf
EAX/AX/AL	11	000	c0	C8	d0	d8	e0	e8	fi)	re
ECX/CX/CL	11	001	c1	C9	d1	d9	e1	e9	f1	f9
EDX/DX/DL	11	010	c2	ca	d2	Da	e2	ea	f2	fa
EBX/BX/BL	11	011	c3	cb	d3	Db	e3	eb	O	ft
ESP/SP/AH	11	100	c4	cc	d4	DC	e4	ec	f4	fc
EBP/BP/CH	11	101	c5	cd	d5	Dd	e5	ed	f5	fd
ESI/SI/DH	И	110	c6	ce	d6	De	e6	ee	f6	fe
EDI/DI/BH	11	111	c7	cf	d7	Df	e7	ef	f7	ff

Рассмотрим пример использования данных таблиц при значении байта mod r/m равном 87h. Для восстановления местонахождения операндов данной машинной команды следует найти это значение в одной из таблиц (какой таблицей воспользоваться, зависит от текущего режима адресации) и по первому столбцу строки, содержащей код 87h, определить местонахождение *первого* операнда. В нашем случае адрес операнда формируется из содержимого регистра BX и 16-разрядного смещения, значение которого следует искать в следующих за байтом mod r/m двух байтах. Второй столбец той же строки содержит значение поля mod. Третье поле байта mod r/m можно найти, переместившись вверх по столбцу, содержащему значение 87h, до пересечения со строкой reg или /цифра. При этом будет выбрано значение, идентифицирующее один из регистров или продолжение кода операции. Что именно — определяется либо самим кодом операции, либо значением бита w в сочетании с текущей разрядностью адреса (см. ранее).

При использовании 32-разрядной адресации содержимое байта mod r/m трактуется несколько иначе из-за наличия в формате машинной команды байта sib (см. подраздел «Байт масштаба, индекса и базы»).

Некоторые машинные команды могут работать с сегментными регистрами. Далее приведены соглашения по кодированию сегментных регистров. В дальнейшем изложении будем различать два набора регистров:

- sreg86 — сегментные регистры, существовавшие в архитектуре процессоров i8086/88 и i80286;
- sreg386 — сегментные регистры архитектуры процессоров i80386 и выше.

Различие наборов состоит в том, что кодируются они различным количеством битов: sreg86 — двумя битами (табл. 3.7), а sreg386 — тремя (табл. 3.8).

Таблица 3.7. Кодировка сегментных регистров в наборе sreg86

Код в поле sreg86	Сегментный регистр
00	ES
01	CS
10	SS
11	DS

Таблица 3.8. Кодировка сегментных регистров в наборе sreg386

Код в поле sreg386	Сегментный регистр
000	ES
001	CS
010	SS
011	DS
100	FS
101 GS	

Одна из целочисленных команд — команда MOV — может оперировать системными регистрами. Кодировка этих регистров приведена в табл. 3.9.

Таблица 3.9. Кодировка системных регистров в команде MOV

Код в поле creg	Регистры управления	Регистры отладки
000	CRO	DRO
001	—	DR1
010	CR2	DR2
011	CR3	DR3
100	CR4	—
101	—	—
HO	—	DR6
111	—	DR7

Байт масштаба, индекса и базы

Байт масштаба, индекса и базы (Scale-Index-Base — sib) используется для расширения возможностей адресации операндов. На наличие байта sib в машинной команде указывает сочетание одного из значений 01 или 10 поля mod и значения поля r/m = 100. Байт sib состоит из трех элементов (табл. 3.10).

- В поле *масштаба* (ss) размещается *масштабный множитель* для индексного компонента index, занимающего следующие три бита байта sib. В поле ss может содержаться значение 1, 2, 4 или 8. При вычислении эффективного адреса на это значение будет умножаться содержимое индексного регистра. Более подробно, с практической точки зрения, эта расширенная возможность индексации рассматривается при обсуждении массивов в главе 13.
- Поле index позволяет хранить номер индексного регистра, содержимое которого применяется для вычисления эффективного адреса операнда.
- Поле base требуется для хранения номера базового регистра, содержимое которого также применяется для вычисления эффективного адреса операнда. В качестве базового и индексного регистров могут использоваться большинство регистров общего назначения.

Таблица 3.10. Значения байта sib (32-разрядная адресация)

r32			EAX	ECX	EDX	EBX	ESP	*	ESI	EDI
База (base)			000	001	010	011	100	101	110	111
Масштабирование индексного регистра	Масштабный множитель (ss)	Индекс (index)	**Шестнадцатеричные** значения sib							
[EAX]	00	000	00	01	02	03	04	05	06	07
[ECX]	00	001	08	09	0A	0B	0C	0D	0E	0F
[EDX]	00	010	10	11	12	13	14	15	16	17
[EBX]	00	011	18	19	1A	1B	1C	1D	1E	1F
—	00	100	20	21	22	23	24	25	26	27
[EBP]	00	101	28	29	2A	2B	2C	2D	2E	2F
[ESI]	00	ПО	30	31	32	33	34	35	36	37
[EDI]	00	111	38	39	3A	3B	3C	3D	3E	3F
[EAX*2]	01	000	40	41	42	43	44	45	46	47
[ECX*2]	01	001	48	49	4A	4B	4C	4D	4E	4F
[EDX*2]	01	010	50	51	52	53	54	55	56	57
[EBX*2]	01	011	58	59	5A	5B	5C	5D	5E	5F
—	01	100	60	61	62	63	64	65	66	67
[EBP*2]	01	101	68	69	6A	6B	6C	6D	6E	6F
[ESI*2]	01	110	70	71	72	73	74	75	76	77
[EDI*2]	01	111	78	79	7A	7B	7C	7D	7E	7F
[EAX*4]	10	000	80	81	82	83	84	85	86	87

Масштабирование индексного регистра	Масштабный множитель (SS)	Индекс (index)	**Шестнадцатеричные** значения sib							
[ECX*4]	10	001	88	89	8A	8B	8C	8D	8E	8F
[EDX*4]	10	010	90	91	92	93	94	95	96	97
[EBX*4]	10	011	98	99	9A	9B	9C	9D	9E	9F
—	10	100	AO	A1	A2	A3	A4	A5	A6	A7
[EBP*4]	10	101	A8	A9	AA	AB	AC	AD	AE	AF
[ES1*4]	10	ПО	BO	B1	B2	B3	B4	B5	B6	B7
[EDI*4]	10	111	B8	B9	BA	BB	BC	BD	BE	BF
[EAX*8]	И	000	CO	C1	C2	C3	C4	C5	C6	C7
[ECX*8]	И	001	C8	C9	CA	CB	CC	CD	CE	CF
[EDX*8]	И	010	D0	D1	D2	D3	D4	D5	D6	D7
[EBX*8]	И	on	D8	D9	DA	DB	DC	DD	DE	DF
—	11	100	EO	E1	E2	E3	E4	E5	E6	E7
[EBP*8]	И	101	E8	E9	EA	EB	EC	ED	EE	EF
[ESI*8]	11	ПО	FO	F1	F2	F3	F4	F5	F6	F7
[EDI*8]	11	111	F8	F9	FA	FB	FC	FD	FE	FF

По значению байта sib легко восстановить компоненты машинных команд, содержащие адрес операнда с учетом его масштабирования.

Одно значение базового регистра (base) в табл. 3.10 замещено символом звездочки (*). Это означает наличие в команде адреса смещ_32 без базы, если mod равно 00, и [EBP] — в противном случае. Такой подход обеспечивает следующие режимы адресации:

- смещ_32[индекс], если mod=00;
- смещ_8[ebp][индекс], если mod=01;
- смещ_32[ebp][индекс], если mod=10.

Поля смещения и непосредственного операнда

Поле смещения в команде — это 8-, 16- или 32-разрядное целое число со знаком, представляющее собой полностью или частично (с учетом приведенных ранее рассуждений) значение эффективного адреса операнда.

Поле непосредственного операнда — необязательное поле, представляющее собой 8-, 16- или 32-разрядный непосредственный операнд. Наличие этого поля, конечно, отражается на значении байта mod r/m.

Столь подробное обсуждение различных полей машинной команды, в том числе с использованием всех приведенных ранее таблиц, имеет целью показать правила формирования операндов машинных команд. При рассмотрении синтаксиса ассемблера (глава 5) на основе этого материала будут обсуждаться правила записи операндов команд ассемблера.

Функциональная классификация машинных команд

В начале главы отмечалось, что система команд последнего на сегодняшний день процессора Pentium IV архитектуры IA-32 содержит более 300 машинных команд. Весь набор машинных команд можно разбить на четыре группы (рис. 3.2).



Рис. 3.2. Машинные команды процессора Intel (Pentium IV)

В пределах каждой из этих больших групп, исходя из функционального назначения отдельных команд, можно провести дальнейшее разбиение на более мелкие подгруппы. Такой подход позволяет достичь нескольких целей:

- оценить возможности процессора по обработке данных;
- рассмотреть совокупность команд процессора архитектуры IA-32 как иерархическую и самодостаточную систему;
- осмысленно изучать отдельные машинные команды в контексте остальных.

В главах 7-10 и 12 будут рассмотрены команды основного процессора (целочисленные), поэтому здесь приведем их классификацию по функциональному признаку (рис. 3.3).

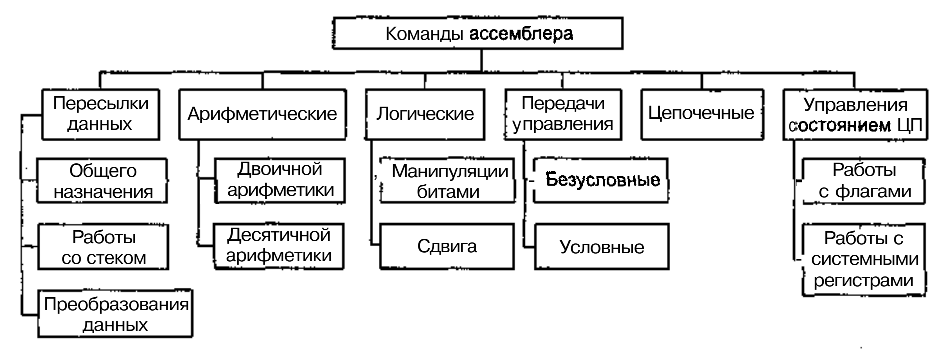


Рис. 3.3. Функциональная классификация целочисленных машинных команд

Функциональная классификация команд остальных групп будет приведена в материале соответствующих глав. Информацию о каждой команде процессора архитектуры IA-32 (вплоть до Pentium IV) можно найти в приложении, которое удобно использовать в ходе работы над программными проектами. Стоит обратить внимание на то, что приложение состоит из четырех частей — это соответствует разбиению команд процессора на функциональные группы (рис. 3.2).

Итоги

- Система машинных команд — важнейшая часть архитектуры компьютера, определяющая возможности его программирования.
- Для работы процессора достаточно программы в двоичных кодах, но такое прямое программирование на практике не используется. Язык ассемблера — символический аналог машинного языка. Преобразование команд ассемблера в соответствующие машинные команды производит программа-транслятор — ассемблер. Дальнейшая интерпретация машинных команд в конкретные сигналы электронных схем осуществляется с помощью блока микропрограммного управления, входящего в состав процессора.
- Существует взаимно однозначное соответствие машинных команд и команд ассемблера.
- Кодирование машинных команд производится в соответствии с одним из возможных форматов. Команды процессоров архитектуры IA-32 считаются сложными, так как в основу концепции процессоров Intel положен принцип совместимости — программы, разработанные для более ранних моделей процессоров, должны выполняться на всех последующих.
- Структура команд процессора позволяет обеспечить большую гибкость при обработке операндов и разнообразие режимов адресации.
- Большинство команд ассемблера имеют несколько возможных вариантов сочетания операндов. Для каждого возможного сочетания операндов имеется своя машинная команда со своим значением поля кода операции.
- Машинную команду формируют несколько полей, из которых обязательным является только одно — поле кода операции.
- Система команд архитектуры IA-32 является иерархической и самодостаточной. Все команды делятся на четыре группы, в пределах каждой из которых выделяется довольно большое количество функциональных подгрупп.

Глава 4
Системы счисления

- **Позиционные и непозиционные системы счисления**
- **Двоичная система счисления**
- **Шестнадцатеричная система счисления**
- **Десятичная система счисления**
- **Перевод чисел из одной системы счисления в другую**

Понимание порядка представления чисел в двоичной, десятичной и шестнадцатеричной системах счисления является одним из необходимых условий успешного программирования на ассемблере. Не менее важно уметь использовать правила конвертирования числовых данных между этими системами счисления. Причинами этого являются, с одной стороны, двоичная организация современных ЭВМ и, с другой стороны, более привычная для человека работа с числовой информацией на основе десятичной системы счисления.

Как известно, *системой счисления* называется совокупность правил записи чисел. Системы счисления подразделяются на *позиционные* и *непозиционные*. Как в позиционных, так и в непозиционных системах счисления используется определенный набор символов — *цифр,* последовательное сочетание которых образует число. Непозиционные системы счисления появились раньше позиционных. Они характеризуются тем, что в них символы, обозначающие то или иное число (то есть цифры), не меняют своего значения в зависимости от местоположения в записи этого числа. Классическим примером такой системы счисления является *римская.* В ней для записи чисел используются буквы латинского алфавита. При этом буква I означает единицу, V — пять, X — десять, L — пятьдесят, C — сто, D — пятьсот, M — тысячу. Для получения количественного эквивалента числа в римской системе счисления необходимо просто просуммировать количественные эквиваленты входящих в него цифр. Исключение из этого правила составляет случай, когда младшая цифра находится перед старшей, — в этом случае нужно не складывать, а вычитать число вхождений этой младшей цифры. Например:

DLXXVII = 500 + 50 + 10 + 10 + 5 + 1 + 1 = 577.

Другой пример:

CDXXIX = 500 - 100 + 10 + 10 - 1 + 10 = 429.

В позиционной системе счисления количество символов в наборе равно *основанию* системы счисления. Место каждой цифры в числе называется *позицией.* Номер позиции символа (за вычетом единицы) в числе называется *разрядом.* Разряд 0 называется *младшим* разрядом. Каждой цифре соответствует определенный количественный эквивалент. Введем обозначение — запись $A_{(p)}$ будет означать количественный эквивалент числа А, состоящего из *n* цифр a_k (где $k = 0, ..., n - 1$) в системе счисления с основанием *p*. Это число можно представить в виде последовательности цифр:

$$A_{(p)} - an_{-1}an_{-2} \dots a_1a_0.$$

При этом, конечно, всегда выполняется неравенство $a_k < p$.

В общем случае количественный эквивалент некоторого положительного числа А в позиционной системе счисления можно представить выражением:

$$A_{(p)} = a_{n-1} \cdot p^{n-1} + a_{n-2} \cdot p^{n-2} + \dots + a_1 \cdot p^1 + a_0 \cdot p^0, \qquad (4.1)$$

где *p* — основание системы счисления (некоторое целое положительное число), *a* — цифра данной системы счисления, *n* — номер старшего разряда числа.

Для получения количественного эквивалента числа в некоторой позиционной системе счисления необходимо сложить произведения количественных значений цифр на степени основания, показатели которых равны номерам разрядов (обратите внимание на то, что нумерация разрядов начинается с нуля).

После такого формального введения можно приступить к обсуждению некоторых позиционных систем счисления, наиболее часто используемых при разработке программ на ассемблере.

Двоичная система счисления

Набор цифр для двоичной системы счисления — {0, 1}, основание степени *(p)* — 2.

Количественный эквивалент некоторого целого *n*-значного двоичного числа вычисляется согласно формуле (4.1):

$$A_{(2)} - a_{n-1} \cdot 2^{n-1} + a_{n-2} \cdot 2^{n-2} + \dots + a_1 \cdot 2^1 + V\, 2^\circ. \qquad (4.2)$$

Как мы уже отмечали, наличие этой системы счисления обусловлено тем, что компьютер построен на логических схемах, имеющих в своем элементарном виде только два состояния — включено и выключено. Производить счет в двоичной системе просто для компьютера, но сложно для человека. Например, рассмотрим двоичное число 10100111.

Вычислим десятичный эквивалент этого двоичного числа. Согласно формуле (4.2), это будет величина, равная следующей сумме:

$$1 \cdot T + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^\circ.$$

Посчитайте сами, сколько получится.

Сложение и вычитание двоичных чисел (рис. 4.1) выполняется так же, как и в других позиционных системах счисления, например десятичной. Точно так же выполняется заем (перенос) единицы из старшего разряда (в старший разряд).

```
  11   11111    перенос      1  1   1      заем
   110011011               11010010011
 + 110010101             - 00111011011
  ----------              ------------
  1100110000               10010111000
```

Рис. 4.1. Сложение и вычитание двоичных чисел

Приведем степени двойки (табл. 4.1).

Таблица 4.1. Степени двойки

Степень	Два в указанной степени
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024
11	2048
12	4096

Шестнадцатеричная система счисления

Шестнадцатеричная система счисления имеет набор цифр {0, 1, 2, ..., 9, A, B, C, D, E, F} и основание степени (p)— 16.

Количественный эквивалент некоторого целого n-значного шестнадцатеричного числа вычисляется согласно формуле (4.1):

$$A_{(16)} = a_{n-1} \cdot 16^{n-1} + a_{n-2} \cdot 16^{n-2} + \ldots + a_1 \cdot 16^1 + a_0 \cdot 16^0.$$

К примеру, количественный эквивалент шестнадцатеричного числа f45ed23c равен:

$$15 \cdot 16^7 + 4 \cdot 16^6 + 5 \cdot 16^5 + 14 \cdot 16^4 + 13 \cdot 16^3 + 2 \cdot 16^2 + 3 \cdot 16^1 + 12 \cdot 16^0.$$

Посчитайте сами, сколько получится. Приведем соответствие двоичных чисел и их десятичных и шестнадцатеричных эквивалентов (табл. 4.2).

Таблица 4.2. Шестнадцатеричные цифры

Десятичное число	Двоичная тетрада	Шестнадцатеричное число
0	0000	0
1	0001	1
2	0010	2

Десятичное число	Двоичная тетрада	Шестнадцатеричное число
3	ООП	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A, a
11	1011	B, b
12	1100	C, c
13	1101	D, d
14	1110	E, e
15	1111	F, f
16	10000	10

Поначалу запомнить эти соотношения сложно, поэтому полезно иметь под руками некоторую справочную информацию. Табл. 4.2 содержит представления десятичных чисел из диапазона 0-16 в двоичной и шестнадцатеричной системах счисления. Ее удобно использовать для взаимного преобразования чисел в рассматриваемых нами трех системах счисления. Шестнадцатеричная система счисления при вычислениях несколько сложнее, чем двоичная, в частности, в том, что касается правил переносов в старшие разряды (заемов из старших разрядов). Главное здесь — помнить следующее равенство:

$$<1 + F = 10)_{16}.$$

Эти переходы очень важны при выполнении сложения и вычитания шестнадцатеричных чисел (рис. 4.2).

```
  1 1          перенос          1 1     заем
   E F 1 5   1 слагаемое        B C D 8  уменьшаемое
+  C 1 E 8   2 слагаемое      - 5 E F 4  вычитаемое
 ----------                     --------
 1 B 0 F D    результат         5 D E 4  результат
```

Рис. 4.2. Сложение и вычитание шестнадцатеричных чисел

Десятичная система счисления

Десятичная система счисления наиболее известна, так как она постоянно используется нами в повседневной жизни. Данная система счисления имеет набор цифр {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} и основание степени (p)- 10.

Количественный эквивалент некоторого целого n-значного десятичного числа вычисляется согласно формуле (4.1):

$$A_{(10)} - a_{n-1} \cdot 10^{n-1} + a_{n-2} \cdot 10^{n-2} + \ldots + a_1 \cdot 10^1 + a_0 \cdot 10^0.$$

К примеру, значение числа $A_{(10)} = 4523$ равно:

$$4 \cdot 10^3 + 5 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0.$$

Перевод чисел из одной системы счисления в другую

Одного знания о существовании разных систем счисления мало. Для того чтобы в полной мере использовать их в своей практической работе при программировании на ассемблере, необходимо научиться выполнять взаимное преобразование чисел между тремя системами счисления. Этим мы и займемся в дальнейшем. Кроме того, дополнительно будут рассмотрены некоторые особенности процессоров Intel при работе с числами со знаком.

Перевод в десятичную систему счисления

Перевод в десятичную систему счисления является самым простым. Обычно его производят с помощью так называемого *алгоритма замещения,* суть которого заключается в следующем: сначала в десятичную систему счисления переводится основание степени *р,* а затем — цифры исходного числа. Результаты подставляются в формулу (4.1). Полученная сумма и будет искомым результатом. Неявно при обсуждении двоичной и шестнадцатеричной систем счисления мы производили как раз такое преобразование.

Перевод в двоичную систему счисления

Перевод из десятичной системы счисления

Перевод числа в двоичную систему счисления из десятичной выполняется по описанному далее алгоритму.

1. Разделить десятичное число А на 2. Запомнить частное *q* и остаток *a.*
2. Если в результате шага 1 частное *q* не равно 0, то принять его за новое делимое и отметить остаток *a,* который будет очередной значащей цифрой числа, и вернуться к шагу 1, на котором в качестве делимого (десятичного числа) участвует полученное на шаге 2 частное.
3. Если в результате шага 1 частное *q* равно 0, алгоритм прекращается. Выписать остатки в порядке, обратном их получению. Получится двоичный эквивалент исходного числа.

К примеру, перевод в двоичную систему счисления числа 247_{10} иллюстрирует рис. 4.3. Порядок обхода остатков для получения результата (11110111_2) показан стрелками.

Перевод из шестнадцатеричной системы счисления

Перевод из шестнадцатеричной системы счисления мы уже обсуждали ранее. Суть его заключается в последовательной замене шестнадцатеричных цифр соответствующими двоичными тетрадами, согласно табл. 4.2. К примеру, двоичное число, соответствующее числу $e4d5_{16}$, равно $1110\,0100\,1101\,0101_2$.

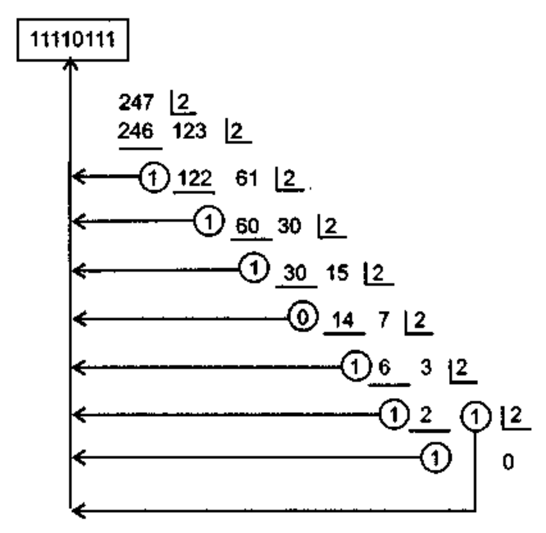


Рис. 4.3. Перевод в двоичную систему счисления

Перевод в шестнадцатеричную систему счисления

Перевод из десятичной системы счисления

Общая идея алгоритма перевода из десятичной системы счисления в шестнадцатеричную аналогична рассмотренной ранее в алгоритме перевода в двоичную систему счисления из десятичной.

1. Разделить десятичное число А на 16. Запомнить частное *q* и остаток *a*.
2. Если в результате шага 1 частное *q* не равно 0, то принять его за новое делимое, записать остаток и вернуться к шагу 1.
3. Если частное *q* равно 0, прекратить работу алгоритма. Выписать остатки в порядке, обратном их получению. Получится шестнадцатеричный эквивалент исходного десятичного числа.

К примеру, перевод в шестнадцатеричную систему счисления числа $32\,767_{10}$ иллюстрирует рис. 4.4. Порядок обхода остатков для получения результата ($7fff_{16}$) показан стрелками.

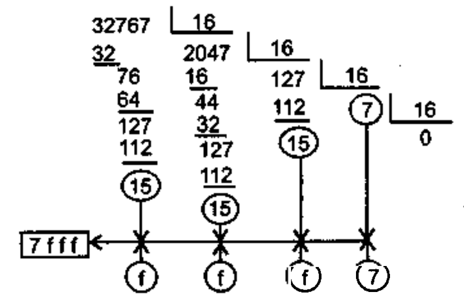


Рис. 4.4. Перевод в шестнадцатеричную систему счисления

Перевод из двоичной системы счисления

Идея алгоритма состоит в том, что двоичное число разбивается на тетрады начиная с младшего разряда. Далее каждая тетрада приводится к соответствующему шестнадцатеричному числу, согласно табл. 4.2.

К примеру, пусть требуется перевести в шестнадцатеричную систему счисления следующее число:

$$1110010110101111010110001101100011110101010110 1_2.$$

Разобьем его на тетрады:

0111 0010 1101 0111 1010 11000110 11000111 1010 1010 1101.

По тетрадам приводим последовательности нулей и единиц к шестнадцатеричному представлению:

7 2 d 7 a c 6 c 7 a a d.

То есть в результате преобразования мы получим шестнадцатеричное число

$$72d7ac6c7aad_{16}.$$

Перевод дробных чисел

Программист должен уметь выполнять перевод в различные системы счисления не только целых чисел, но и дробных чисел. Особенно важно это при программировании алгоритмов, использующих операции с плавающей точкой. Без владения в полной мере знаниями о том, как представляются дробные числа в памяти компьютера и в регистрах сопроцессора, вам вряд ли удастся овладеть программированием на ассемблере в полной мере. Давайте разберемся с наиболее часто используемыми на практике способами перевода дробных чисел. Для этого формулу (4.1) преобразуем к следующему виду:

$$A_{(p)} = a_{n-1} \cdot p^{n-1} + a_{n-2} \cdot p^{n-2} + \ldots$$
$$\ldots + a_1 \cdot p^1 + a_0 \cdot p^0 + a_{-1} \cdot p^{-1} + a_{-2} \cdot p^{-2} + \ldots + a_{-m} \cdot p^{-m}. \quad (4.3)$$

Рассмотрим операции перевода чисел на примерах.

Пример 1

Пусть требуется перевести в десятичное представление следующее дробное число, заданное в двоичной системе счисления:

$$110100{,}01001011_2.$$

Для перевода используем формулу (4.3):

$$110100{,}01001011_2 = 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} +$$
$$+ 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 0 \cdot 2^{-4} + 1 \cdot 2^{-5} + 0 \cdot 2^{-6} + 1 \cdot 2^{-7} + 1 \cdot 2^{-8}.$$

Для вас не составит труда вычислить целую часть десятичной дроби:

$$1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0.$$

Для вычисления остальной части выражения удобно использовать табл. 4.3.

Таблица 4.3. Значения отрицательных степеней по основанию числа 2

Отрицательная степень	Два в указанной степени
1	0,5
2	0,25
3	0,125
4	0,0625
5	0,03125
6	0,015625
.7	0,0078125

Вы можете сами продолжить табл. 4.3 значениями отрицательных степеней по основанию числа 2 и в конце концов подсчитать результат перевода в десятичное представление значения

$$110100{,}01001011_2.$$

Пример 2

Пусть требуется перевести в десятичное представление следующую дробь, заданную в шестнадцатеричной системе счисления:

$$1df2{,}a1e4_{16}.$$

Вновь используем формулу (4.3):

$$1df2{,}a1e4_{16} = 1 \cdot 16^3 + 13 \cdot 16^2 + 15 \cdot 2^1 + 2 \cdot 16'' + 10 \cdot 16^{-1} + 1 \cdot 16^2 +$$
$$+ 14 \cdot 16^{-3} + 4 \cdot 16^{-4}.$$

Для удобства вычисления дробной части приведем значения отрицательных степеней числа 16 (табл. 4.4).

Таблица 4.4. Значения отрицательных степеней по основанию числа 16

Отрицательная степень	Шестнадцать в указанной степени
1	0,0625
2	0,00390625
3	0,000244140625
4	0,0000152587890625
5	0,00000095367431640625
6	0,000000059604644775390625
7	0,0000000037252902984619140625

Перевод из двоичной системы счисления в шестнадцатеричную и обратно выполняется, как обычно, на основе тетрад и трудности вызывать не должен.

Пример 3

Рассмотрим теперь проблему представления десятичных дробей в двоичной и шестнадцатеричной системах счисления:

Общий алгоритм перевода десятичной дроби в другую систему счисления можно представитьследующейпоследовательностьюшагов.

1. Выделить целую часть десятичной дроби и выполнить ее перевод в выбранную систему счисления по алгоритмам, рассмотренным ранее.
2. Выделить дробную часть и умножить ее на основание выбранной новой системы счисления.
3. В полученной после умножения дробной части десятичной дроби выделить целую часть и принять ее в качестве значения первого после запятой разряда числа в новой системе счисления.
4. Если дробная часть значения, полученного после умножения, равна нулю, то прекратить процесс перевода. Процесс перевода можно прекратить также в случае, если достигнута необходимая точность вычисления. В противном случае вернуться к шагу 3.

Рассмотрим пример. Пусть требуется перевести в двоичную систему счисления десятичную дробь $108{,}406_{10}$.

Сначала переведем целую часть десятичной дроби $108{,}406_{10}$ в двоичную систему счисления (рис. 4.5).

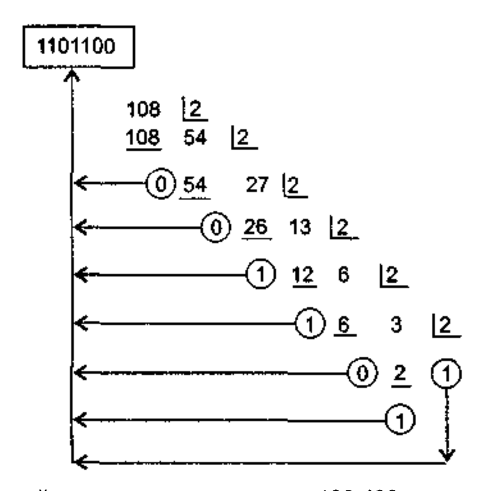


Рис. 4.5. Перевод целой части десятичного числа 108,406 в двоичную систему счисления

Затем переведем дробную часть десятичного числа $108{,}406_{10}$ (рис. 4.6) по приведенному ранее алгоритму.

Результат перевода следующий:

$$108{,}406_{10} = 1101100{,}011001111.$$

При переводе дробного числа из десятичной системы счисления в шестнадцатеричную целесообразно предварительно перевести число в двоичную! систему, а затем двоичное представление разбить на тетрады отдельно до разделительной

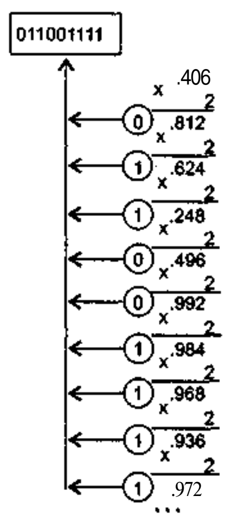


Рис. 4.6. Перевод дробной части числа 108,406 в двоичную систему счисления

запятой и после запятой. Разбиение на тетрады двоичных разрядов целой части производят от запятой в сторону старших разрядов. Неполную старшую тетраду дополняют слева нулями. Разряды дробной части, напротив, разбивают на тетрады от запятой вправо к младшим разрядам. Если последняя тетрада неполная, то ее дополняют нулями справа. На рис. 4.7 показан процесс перевода того же дробного десятичного числа ($108,406_{10}$) в эквивалентное шестнадцатеричное представление.

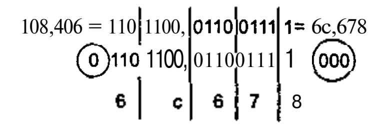


Рис. 4.7. Пример перевода десятичного числа в шестнадцатеричную систему счисления

Перевод чисел со знаком

До сих пор предполагалось, что числа положительные. А как представляются в компьютере числа со знаком?

Положительные целые со знаком — это 0 и все положительные числа.

Отрицательные целые со знаком — это все числа, меньшие 0. Отличительным признаком числа со знаком является особая трактовка старшего бита поля, представляющего число. В качестве поля могут выступать байт, слово или двойное слово. Естественно, что физически этот бит ничем не отличается от других — все зависит от команды, работающей с данным полем. Если в ее алгоритме заложена возможность работы с целыми числами со знаком, то команда будет по-особому трактовать старший бит поля. Если старший бит равен 0, число считается положительным и его значение вычисляется по правилам, которые мы рассмотрели ранее. Если старший бит равен 1, число считается отрицательным, а это предполагает, что оно записано в так называемом *дополнительном коде.* Разберемся в том, что он собой представляет.

Дополнительный код некоторого отрицательного числа представляет собой результат инвертирования (замены 1 на 0 и наоборот) каждого бита двоичного числа, равного модулю" исходного отрицательного числа плюс единица. К примеру, рассмотрим десятичное число -185_{10}. Модуль данного числа в двоичном представлении равен $1011\,1001_2$. Сначала нужно дополнить это значение слева нулями до нужной размерности — байта, слова и т. д. В нашем случае дополнить нужно до слова, так как диапазон представления знаковых чисел в байте составляет -128...127. Следующее действие — получить *двоичное дополнение*. Для этого все разряды двоичного числа нужно инвертировать:

$$000000001011\,1001_2 \rightarrow 111\,111\,1101000110_2.$$

Теперь прибавляем единицу:

$$1111111\,101000110_2 + 0000000000000001_2 = 1111111101000111_2.$$

Результат преобразования равен 1111111101000111_2. Именно так и представляется число -185_{10} в компьютере.

При работе с числами со знаком от вас наверняка потребуется умение выполнять обратное действие — имея двоичное дополнение числа, определить значение его модуля. Для этого необходимо выполнить два действия.

1. Выполнить инвертирование битов двоичного дополнения.
2. К полученному двоичному числу прибавить двоичную единицу.

К примеру, определим модуль двоичного представления числа

$$-185_{10} = 1111111101000111_2.$$

Сначала инвертируем биты:

$$\text{ИНН}\ 1101000111_2 \rightarrow 00000000101\,11000_2.$$

Добавляем двоичную единицу:

$$0000000010111000_2 + 0000000000000001_2 = 00000000101\,11001_2 = |-185|.$$

Теперь мы готовы разговаривать с компьютером на его языке, состоящем из команд и данных. В следующей главе мы напишем первую программу на ассемблере, разбирая которую, мы сможем применить знания, полученные в этой и предыдущих главах.

Итоги

ж Системы счисления подразделяются на позиционные и непозиционные. Как в позиционных, так и в непозиционных системах счисления используется определенный набор символов — цифр, последовательное сочетание которых образует число.

и Специфика работы программиста предполагает хорошее владение счетом в трех системах счисления: двоичной, шестнадцатеричной и десятичной. При этом программист должен достаточно уверенно переводить числа из одной системы счисления в другую.

т Числа со знаком представляются в компьютере особым образом: положительные числа — в виде обычного двоичного числа, а отрицательные — в дополнительном коде.

Глава 5
Синтаксис ассемблера

- Структура программы на ассемблере
- Типы и структура предложений ассемблера
- Понятие о метасинтаксических языках
- Классификация лексем ассемблера
- Описание простых операндов и операндов-выражений
- Варианты расположения операндов команд ассемблера
- Виды адресации операндов в памяти
- Операторы ассемблера
- Стандартные директивы сегментации
- Упрощенные директивы сегментации
- Простые типы данных ассемблера (диапазоны значений)
- Директивы описания простых типов данных

В предыдущих главах основное обсуждение было посвящено внутреннему устройству процессора, его принципам работы и программной модели. И это не случайность — чем более низкий уровень функционирования компьютера доступен пониманию программиста, тем легче и осмысленнее для него становится процесс изучения и дальнейшего программирования на языке ассемблера. Сам язык ассемблера пока обсуждался мало. В основном речь шла о нем как о символическом аналоге машинного языка. В связи с этим отмечалось, что программа, написанная на ассемблере, отражает основные особенности архитектуры процессора: организацию памяти, способы адресации операндов, правила использования регистров и т. д. Также говорилось, что необходимость учета подобных особенностей делает

ассемблер уникальным для каждого типа процессоров. Эта и следующие за ней главы будут посвящены изучению правил оформления и разработки программ на языке ассемблера с учетом влияния на эти правила архитектуры IA-32.

Синтаксис ассемблера

Программа на ассемблере представляет собой совокупность блоков памяти, называемых *сегментами.* Программа может состоять из одного или нескольких таких блоков-сегментов. Сегменты программы имеют определенное назначение, соответствующее типу сегментов: кода, данных и стека. Названия типов сегментов отражают их назначение. Деление программы на сегменты отражает сегментную организацию памяти процессоров Intel (архитектура IA-32). Каждый сегмент состоит из совокупности отдельных строк, в терминах теории компиляции называемых *предложениями* языка. Для языка ассемблера предложения, составляющие программу, могут представлять собой синтаксические конструкции четырех типов.

- *Команды* (инструкции) представляют собой символические аналоги машинных команд. В процессе трансляции инструкции ассемблера преобразуются в соответствующие команды системы команд процессора.
- *Макрокоманды* — это оформляемые определенным образом предложения текста программы, замещаемые во время трансляции другими предложениями.
- *Директивы* являются указанием транслятору ассемблера на выполнение некоторых действий. У директив нет аналогов в машинном представлении.
- *Комментарии* содержат любые символы, в том числе и буквы русского алфавита. Комментарии игнорируются транслятором.

Для распознавания транслятором ассемблера этих предложений их нужно формировать по определенным синтаксическим правилам. Для формального описания синтаксиса языков программирования используются различные *метасинтаксические* языки, которые представляют собой совокупность условных знаков, образующих нотацию метасинтаксического языка, и правил формирования из этих знаков однозначных описаний синтаксических конструкций целевого языка.

В учебных целях удобно использовать два метасинтаксических языка — *синтаксические диаграммы* и *нормальные формы Бэкуса-Наура.* Оба этих языка, в конечном итоге, предоставляют одинаковый объем информации. Поэтому выбор конкретного языка может определяться исходя из того, что синтаксические диаграммы более наглядны, а расширенные формы Бэкуса-Наура более компактны. В учебнике будут использоваться оба способа.

На рис. 5.1, 5.2 и 5.3 показан порядок написания предложений ассемблера с помощью синтаксических диаграмм.

Как использовать синтаксические диаграммы? Очень просто: для этого нужно всего лишь найти и затем пройти путь от входа диаграммы (слева) к ее выходу (направо). Если такой путь существует, то предложение или конструкция являются синтаксически правильными. Если такого пути нет, значит, эту конструкцию компилятор не примет. Иногда на линиях в синтаксических диаграммах присутствуют стрелки. Они говорят о том, что необходимо обратить внимание на направление обхода, указываемое этими стрелками, так как среди путей могут быть

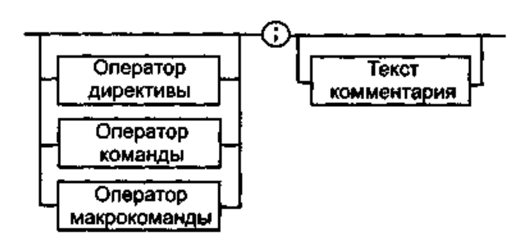


Рис. 5.1. Формат предложений ассемблера

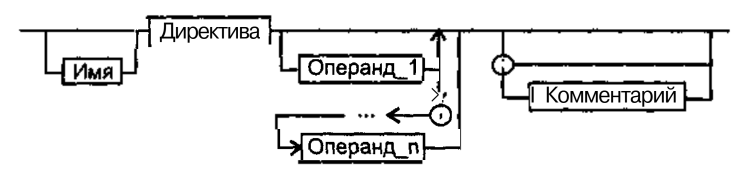


Рис. 5.2. Формат директив

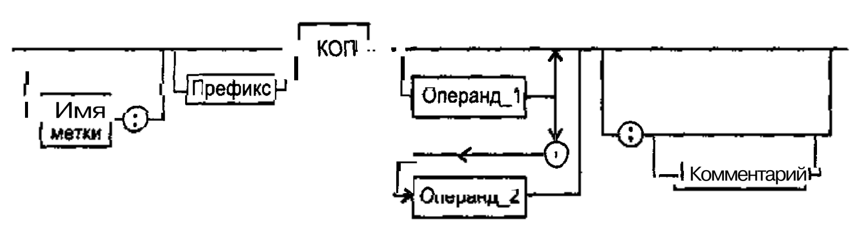


Рис. 5.3. Формат команд и макрокоманд

и такие, по которым можно идти справа налево. По сути, синтаксические диаграммы отражают логику работы транслятора при разборе входных предложений программы. Далее перечислены термины, представленные на рисунках.

- *Имя метки* — символьный идентификатор. Значением данного идентификатора является адрес первого байта предложения программы, которому он предшествует.
- *Префикс* — символическое обозначение элемента машинной команды, предназначенного для изменения стандартного действия следующей за ним команды ассемблера (см. главу 3).
- *Имя* — идентификатор, отличающий данную директиву от других одноименных директив. В зависимости от конкретной директивы в результате обработки ассемблером этому имени могут быть присвоены определенные характеристики.
- *Код операции* (КОП) и *директива* — это мнемонические обозначения соответствующей машинной команды, макрокоманды или директивы транслятора.
- *Операнды* — части команды, макрокоманды или директивы ассемблера, обозначающие объекты, над которыми производятся действия. Операнды ассемблера описываются выражениями с числовыми и текстовыми константами, метками и идентификаторами переменных с использованием знаков операций и некоторых зарезервированных слов.

Другой способ описания синтаксиса языка — *нормальные (расширенные) формы Бэкуса-Наура.* С помощью форм Бэкуса–Наура целевой язык представляется в виде объектов трех типов.

- Основные символы языка, в теории компиляции называемые *терминальными,* — это имена операторов, регистров и т. п., то есть это те символьные объекты, из которых строится, в частности, исходный текст ассемблерной программы.
- Имена конструкций языка, в теории называемые *нетерминальными* символами, обычно заключаются в угловые скобки <> или пишутся строчными буквами.
- Правила (формы) описывают порядок формирования конструкций, в том числе предложений, целевого языка.

Каждая форма состоит из трех частей — левой, правой и связки:

- *левая часть* — всегда нетерминальный символ, который обозначает одну из конструкций языка;
- *связка* — символ стрелки ⇒, который можно трактовать как словосочетание «определяется как»;
- *правая* часть описывает один или несколько вариантов построения конструкции, определяемой *левой* частью.

Несколько форм Бэкуса-Наура могут быть связаны между собой по нетерминальным символам, то есть одна форма определяется через другую. Для построения конструкции целевого языка необходимо взять одну или несколько форм Бэкуса-Наура, в каждой из которых выбрать нужный вариант для подстановки. В конечном итоге должна получиться конструкция (предложение) целевого языка, состоящая только из терминальных символов.

Для примера рассмотрим описание и использование форм Бэкуса-Наура для построения десятичных чисел со знаком. Вначале опишем эти формы (правила):

```
<десятичное_знаковое_целое>⇒<число_без_знака>|+<число_без_знака>|<число_без_знака>
<число_без_знака>⇒<дес_цифра>| <число_без_знака><дес_цифра>
<дес_цифра>⇒0|1|2|3|4|5|6|7|8|9
```

Здесь:

- **<десятичное_знаковое_целое>, <число_без_знака>, <дес_цифра>** — нетерминальные символы (в исходной программе на ассемблере таких объектов нет);
- +|-|0|1|2|3|4|5|6|7|8|9 — терминальные символы (их можно найти в исходном тексте программы), из терминальные символов по приведенным ранее трем правилам строится любое десятичное число;
- символ вертикальной черты (|) означает альтернативу при выборе варианта некоторого правила.

Для примера выведем число -501, используя формы Бэкуса-Наура:

```
<десятичное_знаковое_целое> ⇒ <число_без_знака> ⇒
<число_без_знака><дес_цифра> ⇒ <число_без_знака>1 ⇒
<число_без_знака><дес_цифра>1 ⇒ <число_без_знака>01 ⇒ <дес_цифра>01 ⇒ 501
```

Предложения ассемблера (см. рис. 5.1–5.3) формируются из *лексем,* представляющих собой синтаксически неразделимые последовательности допустимых символов языка, имеющие смысл для транслятора.

Вначале определим алфавит ассемблера, то есть допустимые для написания текста программ символы:

- **ASCII_символ_буква** — все латинские буквы A - Z, a - z, причем прописные и строчные буквы считаются эквивалентными;
- **decdigit** — цифры от 0 до 9;
- специальные знаки _, ?, @, $, &;
- разделители: „ ., [,], (,), <, >, {, }, +, /, *, %, !, ", ", ?, \, = #, ^.

Лексемами языка ассемблера являются ключевые слова, идентификаторы, цепочки символов и целые числа.

Ключевые слова — это служебные символы языка ассемблера. По умолчанию регистр символов ключевых слов не имеет значения. К ключевым словам относятся:

- названия регистров (AL, AH, BL, BH, CL, CH, DL, DH, AX, EAX, BX, EBX, CX, ECX, DX, EDX, BP, EBP, SP, ESP, DI, EDI, SI, ESI, CS, DS, ES, FS, GS, SS, CR0, CR2, CR3, DR0, DR1, DR2, DR3, DR6, DR7);
- операторы (BYTE, SBYTE, WORD, SWORD, DWORD, SDWORD, FWORD, QWORD, TBYTE, REAL4, REAL8, REAL10, NEAR16, NEAR32, FAR16, FAR32, AND, NOT, HIGH, LOW, HIGHWORD, LOWWORD, OFFSET, SEG, LROFFSET, TYPE, THIS, PTR, WIDTH, MASK, SIZE, SIZEOF, LENGTH, LENGTHOF, ST, SHORT, TYPE, OPATTR, MOD, NEAR, FAR, OR, XOR, EQ, NE, LT, LE, GT, GE, SHR, SHL и др.);
- названия команд (КОП) ассемблера, префиксов.

Идентификаторы — последовательности допустимых символов, использующиеся для обозначения имен *переменных* и *меток*. Правило записи идентификаторов можно описать следующими формами Бэкуса–Наура:

```
<id> ⇒ ASCII_символ_буква | ASCII_символ_буква | <id> ASCII_символ_буква |
<id> <decdigit> | <znak> <decdigit> <id> | <znak> <id>
<decdigit>⇒0|1 |2|3|4|5|6|7|8|9
<znak> ⇒ _|?|@|$|_|&
```

Приведенные формы говорят о том, что идентификатор может состоять из одного или нескольких символов. В качестве символов можно использовать буквы латинского алфавита, цифры и некоторые специальные знаки — _, ?, $, @. Идентификатор не может начинаться символом цифры. Длина идентификатора может составлять до 255 символов (247 в MASM), хотя транслятор воспринимает лишь первые 32, а остальные игнорирует. Регулировать длину возможных идентификаторов (в TASM) можно с использованием ключа командной строки /mv. Кроме того, существует возможность указать транслятору на необходимость различать прописные и строчные буквы либо игнорировать их различие (что и делается по умолчанию). Для этого (в TASM) применяются ключи командной строки /mu, /ml, /mx (см. приложение В на сайте http://www.piter.com/download).

Цепочки символов — это последовательности символов, заключенные в одинарные или двойные кавычки. Правила формирования:

```
<string> ⇒ <quote> [[ <stext> ]] <quote>
<stext> ⇒ <StringChar> | <stext> <stringChar>
<stringChar> ⇒ <quote> <quote> | любой_символ_кроме_кавычки
<quote> ⇒ " | '
```

Целые числа могут указываться в двоичной, десятичной или шестнадцатеричной системах счисления. Отождествление чисел при записи их в программах на ассемблере производится по определенным правилам. Десятичные числа не требуют для своего отождествления указания каких-либо дополнительных символов. Для отождествления в исходном тексте программы двоичных и шестнадцатеричных чисел используются следующие правила:

```
<шестнадц_число> ⇒ <дес_шестнадц_число>h | 0<сим_шестнадц_число>h
<дес_шестнадц_число> ⇒ <decdigit><сим_шестнадц_число> | <decdigit>
<сим_шестнадц_число> ⇒
<hexdigit><сим_шестнадц_число>|<дес_шестнадц_число> | <decdigit> | <hexdigit>
<decdigit> ⇒ 0|1 |2 | 3|4|5|6|7|8|9
<hexdigit> ⇒ a | b | c | d | e | f | A | B | C | D | E | F
```

Важно отметить наличие символов после (h) и перед (0) записью шестнадцатеричного числа. Это сделано для того, чтобы транслятор мог отличить в программе одинаковые по форме записи десятичные и шестнадцатеричные числа. К примеру, числа 1578 и 1578h выглядят одинаково, но имеют разные значения. С другой стороны, какое значение в тексте исходной программы может иметь лексема fe023? Это может быть и некоторый идентификатор, и, судя по набору символов, шестнадцатеричное число. Для того чтобы однозначно описать в тексте программы на ассемблере шестнадцатеричное число, начинающееся с буквы, его дополняют ведущим нулем «О» и в конце ставят символ «h». Для данного примера правильная запись шестнадцатеричного числа — Ofe023h:

```
<двоичн_число> ⇒ <bindigit>b|<bindigit><двоичн_число>b
<bindigit> ⇒ 0|1
```

Для двоичных чисел все просто — после записи нулей и единиц, входящих в их состав, необходимо поставить латинскую букву «b». Пример:

```
10010101b
```

Рассуждениями, приведенными ранее, был показан порядок формирования предложений программы ассемблера и составляющих их элементов (лексем). Также были рассмотрены правила формирования меток, названий команд (префиксов). Осталось обсудить комментарии и операнды. Что касается комментария, то это самый простой элемент предложения ассемблера. Любая комбинация символов ASCII, расположенная в строке за символом точки с запятой (;), транслятором игнорируется, то есть является комментарием (см. рис. 5.1-5.3). Описанию операндов, ввиду их важности, будет посвящен отдельный подраздел.

Операнды

Операнды — это объекты, над которыми или при помощи которых выполняются действия, задаваемые инструкциями или директивами. Машинные команды могут либо совсем не иметь операндов, либо иметь один или два операнда. Большинство команд требует двух операндов, один из которых является *источником,* а другой — *приемником (операндом назначения).* В двухоперандной машинной команде возможны следующие сочетания операндов:

- регистр — регистр;
- регистр — память;
- память — регистр;

* непосредственный операнд — регистр;
* непосредственный операнд — память.

Здесь важно подчеркнуть, что один операнд может располагаться в регистре или памяти, а второй операнд *обязательно* должен находиться в регистре или непосредственно в команде. Непосредственный операнд может быть только *источником.*

Для приведенных ранее правил сочетания типов операндов есть исключения, которые касаются:

* *команд работы с цепочками,* которые могут перемещать данные из памяти в память;
* *команд работы со стеком,* которые могут переносить данные из памяти в стек, также находящийся в памяти;
* *команд типа умножения,* которые, кроме операнда, указанного в команде, неявно используют еще и второй операнд.

Операндами могут быть числа, регистры, ячейки памяти, символьные идентификаторы. При необходимости для расчета некоторого значения или определения ячейки памяти, на которую будет воздействовать данная команда или директива, используются *выражения,* то есть комбинации чисел, регистров, ячеек памяти, идентификаторов с арифметическими, логическими, побитовыми и атрибутивными операторами.

Рассмотрим классификацию операндов, поддерживаемых транслятором ассемблера.

* *Операнд задается неявно на микропрограммном уровне.* В этом случае команда явно не содержит операндов. Алгоритм выполнения команды использует некоторые объекты по умолчанию (регистры, флаги в EFLAGS и т. д.). Например, команды CLI и STI неявно работают с флагом прерывания IF в регистре EFLAGS, а команда XLAT неявно обращается к регистру AL и строке в памяти по адресу, определяемому парой регистров DS:BX.
* *Операнд задается в самой команде (непосредственный операнд).* Это может быть число, строка, имя или выражение, имеющее некоторое фиксированное (константное) значение. Физически непосредственный операнд находится в коде команды, то есть является ее частью. Для его хранения в команде выделяется поле длиной до 32 битов (см. главу 3). Непосредственный операнд может быть только вторым операндом (источником). Операнд-приемник может находиться либо в памяти, либо в регистре. Например, команда mov ax,0ffffh пересылает в регистр AX шестнадцатеричную константу 0ffffh. Команда add sum,2 складывает содержимое поля по адресу sum с целым числом 2 и записывает результат по месту первого операнда, то есть в память. Если непосредственный операнд — имя, то оно не должно быть перемещаемым, то есть зависеть от адреса загрузки программы в память. Такое имя можно определить оператором EQU или =. Пример:

```
num equu 5   ; вместо num ассемблер везде подставляет 5
imd = num-2 ; вместо num ассемблер везде подставляет 3
mov al,num  ; эквивалентно mov al,5, здесь 5 - непосредственный операнд
```

```
add [si],imd; сложение [si]:= [si]+3,
              ; здесь imd - непосредственный операнд
mov al,5; al:=5, здесь 5 - непосредственный операнд
```

В данном фрагменте определяются две константы, которые затем используются в качестве непосредственных операндов в командах пересылки MOV и сложения ADD.

- *Адресные операнды* задают физическое расположение операнда в памяти путем указания двух составляющих адреса: сегмента и смещения (рис. 5.4). К примеру:

```
mov ax,0000h
mov ds,ax
mov ax,ds:0000h ; записать слово в ax из области памяти
                ; по физическому адресу 0000:0000
```

Здесь третья команда MOV имеет адресный операнд.

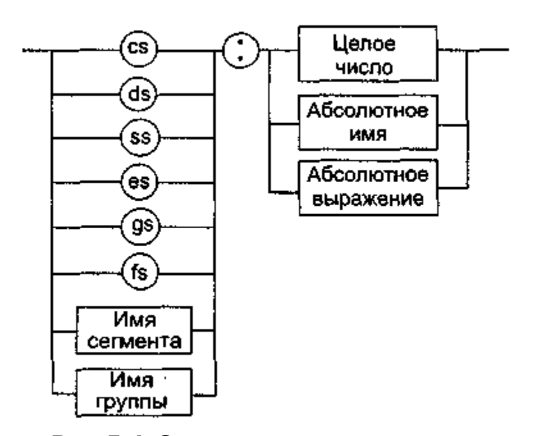


Рис. 5.4. Синтаксис адресных операндов

- *Перемещаемые операнды* — любые символьные имена, представляющие некоторые адреса памяти. Эти адреса могут обозначать местоположение в памяти некоторой инструкции (если операнд — метка) или данных (если операнд — имя области памяти в сегменте данных). Перемещаемые операнды отличаются от адресных тем, что они не привязаны к конкретному адресу физической памяти. Сегментная составляющая адреса перемещаемого операнда неизвестна и определяется после загрузки программы в память для выполнения. К примеру:

```
data segment
mas_w dw 25 dup (0)
...
codesegment
...
lea si,mas_w; mas_w - перемещаемый операнд
```

В этом фрагменте mas_w — символьное имя, значением которого является адрес первого байта области памяти размером 25 слов. Полный физический адрес этой области памяти будет известен только после загрузки программы в память для выполнения.

- *Счетчик адреса* — специфический вид операнда. Он обозначается знаком $. Специфика этого операнда в том, что когда транслятор ассемблера встречает

в исходной программе этот символ, то он подставляет вместо него текущее значение счетчика адреса. Значение счетчика адреса, или, как его иногда называют *счетчика размещения,* представляет собой смещение текущей машинной команды относительно начала сегмента кода. При обработке транслятором очередной команды ассемблера счетчик адреса увеличивается на длину сформированной машинной команды. Важно правильно это понимать. К примеру, обработка директив ассемблера не влечет за собой изменения счетчика, так как директивы ассемблера, в отличие от его **команд,** — это лишь указания транслятору на выполнение определенных действий по формированию машинного представления программы, и для них транслятором не генерируется никаких конструкций в памяти. В качестве примера использования в команде значения счетчика адреса можно привести следующий фрагмент:

```
jmp $+3  ; безусловный переход на команду mov
cld      ; длина команды cld составляет 1 байт
mov al,1
```

При формировании выражения для перехода, подобного $+3, нужно помнить о длине самой команды, в которой это выражение используется, так как значение счетчика адреса соответствует смещению в сегменте команд данной, а не следующей за ней команды. В нашем примере команда **JMP** занимает два байта. Нужно быть осторожным, длина этой и других команд зависит от того, какие в ней используются операнды. Команда с регистровыми операндами будет короче команды, один из операндов которой расположен в памяти. В большинстве случаев эту информацию можно получить, зная формат машинной команды (см. главу 3 и приложение) и анализируя колонку файла листинга с объектным кодом команды.

- *Регистровый операнд* — это просто имя регистра. В программе на ассемблере можно использовать имена всех регистров общего назначения и некоторых системных регистров:
  - 32-разрядные регистры EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP;
  - 16-разрядные регистры AX, BX, CX, DX, **SI, DI,** SP, BP;
  - 8-разрядные регистры AH, AL, BH, BL, CH, CL, DH, DL;
  - сегментные регистры **CS, DS,** SS, ES, FS, GS;
  - системные регистры CR0, CR2, **CR3,** CR4, DR0, DR1, DR2, DR3, DR6, DR7 (см. описание команды MOV в приложении).

  Например, команда add ax,bx складывает содержимое регистров AX и BX и записывает результат в BX. Команда dec si уменьшает содержимое SI на 1. И еще пример:

```
mov al,4        ; константу 4 заносим в регистр al
mov dl,pass+4   ; байт по адресу pass+4 - в регистр dl
add al,dl       ; команда с регистровыми операндами
```

- Операнд — *порт ввода-вывода.* Помимо адресного пространства оперативной памяти процессор поддерживает адресное пространство ввода-вывода, которое используется для доступа к устройствам ввода-вывода. Объем адресного пространства ввода-вывода составляет 64 Кбайт. Для любого устройства компьютера в этом пространстве выделяются адреса. Конкретное значение адреса в пределах этого пространства называется портом ввода-вывода. Физически порту

ввода-вывода соответствует аппаратный регистр (не путать с регистром процессора), доступ к которому осуществляется с помощью специальных команд ассемблера IN и OUT. Например,

```
in  al,60h; ввести байт из порта 60h
```

Регистры, адресуемые с помощью порта ввода-вывода, могут иметь разрядность 8, 16 или 32 бита, но для конкретного порта разрядность регистра фиксирована. Команды IN и OUT работают с фиксированной номенклатурой объектов. В качестве источника информации или получателя применяются так называемые *регистры-аккумуляторы* EAX, AX, AL. Выбор регистра определяется разрядностью порта. Номер порта может задаваться непосредственным операндом в командах IN и OUT или значением в регистре DX. Последний способ позволяет динамически определить номер порта в программе. Например,

```
mov dx,20h   ; записать номер порта 20h в регистр dx
mov al,20h   ; записать значение 20h в регистр al
out dx,al    ; вывести значение 20h в порт 20H
```

- *Структурные операнды* используются для доступа к конкретному элементу сложного типа данных, называемого *структурой.* Мы подробно разберемся со структурами в главе 13.
- *Записи* (аналогично структурному типу) используются для доступа к битовому полю некоторой записи (глава 13).
- *Операнд находится в стеке,*
- *Операнд располагается в памяти.* Это наиболее сложный и в то же время наиболее гибкий способ задания операндов. Он позволяет реализовать *прямой* и *косвенный* варианты адресации, являющиеся основными видами адресации.

Последний вариант расположения операндов, ввиду его важности и большого объема, рассмотрим более подробно. Обсуждение будет сопровождаться примерами команд ассемблера, цель которых — демонстрация того, как изменяется формат команды ассемблера при применении того или иного вида адресации. В связи с этим вернемся еще раз к рис. 2.8 (см. главу 2), который иллюстрирует принцип формирования физического адреса на адресной шине процессора. Видно, что адрес операнда формируется как сумма двух составляющих — сдвинутого на 4 бита содержимого сегментного регистра и 16-разрядного эффективного адреса, который в общем случае вычисляется как сумма трех компонентов: базы, смещения и индекса.

Прямая адресация

Прямая адресация — это простейший вид адресации операнда в памяти, так как эффективный адрес содержится в самой команде и для его формирования не используется никаких дополнительных источников или регистров. Эффективный адрес берется непосредственно из поля смещения машинной команды (см. главу 3), которое может иметь размер 8, 16, 32 бита. Это значение однозначно определяет байт, слово или двойное слово в сегменте данных.

Прямая адресация может быть двух типов.

- *Относительная прямая адресация* используется в командах условных переходов для указания относительного адреса перехода. Относительность такого пе-

рехода заключается в том, что в поле смещения машинной команды содержится 8-, 16- или 32-разрядное значение, которое в результате работы команды будет складываться с содержимым регистра указателя команд IP/EIP. В результате такого сложения получается адрес, по которому и осуществляется переход. К примеру,

```
jc m1   ;переход на метку ml, если флаг cf = 1
mov al,2
...
ml:
```

Хотя в команде указана конкретная метка, ассемблер вычисляет смещение этой метки относительно следующей команды (в нашем случае mov al,2) и подставляет его в формируемую машинную команду JC.

- *Абсолютная прямая адресация* — в этом случае эффективный адрес является частью машинной команды, но формируется этот адрес только из значения поля смещения в команде. Для формирования физического адреса операнда в памяти процессор складывает это поле со сдвинутым на четыре бита значением сегментного регистра. В команде ассемблера можно использовать несколько форм такой адресации. К примеру,

```
mov ax,word ptr [0000]   ;записать слово по адресу
                         ;ds:0000 в регистр ax
```

Однако такая адресация применяется редко — обычно ячейкам в программе присваиваются символические имена. В процессе трансляции ассемблер вычисляет и подставляет значения смещений этих имен в поле смещения формируемой им машинной команды (см. главу 3). В итоге получается, что машинная команда прямо адресует свой операнд, имея, фактически, в одном из своих полей значение эффективного адреса. К примеру,

```
data segment
perl dw 5
...
data ends
code segment
mov ax,data
mov ds,ax
...
mov ax,perl ;записать слово perl (его физический адрес ds:0000) в ax
```

Мы получим тот же результат, что и при использовании команды

```
mov ax,word ptr [0000]
```

Остальные виды адресации относятся к косвенным. Слово «косвенный» в названии этих видов адресации означает, что в самой команде может находиться лишь часть эффективного адреса, а остальные его компоненты находятся в регистрах, на которые указывают своим содержимым байт mod r/m и, возможно, байт sib.

Косвенная адресация имеет следующие разновидности:

- косвенная базовая, или регистровая, адресация;
- косвенная базовая адресация со смещением;
- косвенная индексная адресация со смещением;
- косвенная базовая индексная адресация;
- косвенная базовая индексная адресация со смещением.

Косвенная базовая адресация

При косвенной базовой (регистровой) адресации эффективный адрес операнда может находиться в любом из регистров общего назначения, кроме SP/ESP и BP/EBP (это специальные регистры для работы с сегментом стека).

Синтаксически в команде этот режим адресации выражается заключением имени регистра в квадратные скобки. К примеру, команда mov ax,[ecx] помещает в регистр AX содержимое слова по адресу сегмента данных со смещением, хранящимся в регистре ECX. Так как содержимое регистра легко изменить в ходе работы программы, данный способ адресации позволяет динамически назначить адрес операнда для некоторой машинной команды. Это очень полезно, например, для организации циклических вычислений и для работы с различными структурами данных типа таблиц или массивов.

Косвенная базовая адресация со смещением

Косвенная базовая (регистровая) адресация со смещением является дополнением предыдущего вида адресации и предназначена для доступа к данным с известным смещением относительно некоторого базового адреса. Этот вид адресации удобно использовать для доступа к элементам структур данных, когда смещение элементов известно заранее на стадии разработки программы, а базовый (начальный) адрес структуры должен вычисляться динамически на стадии выполнения программы. Модификация содержимого базового регистра позволяет обращаться к одноименным элементам различных экземпляров однотипных структур данных.

К примеру, команда mov ax,[edx+3h] пересылает в регистр AX слово из области памяти по адресу, определяемому содержимым EDX + 3h. Команда mov ax,mas[dx] пересылает в регистр AX слово по адресу, определяемому содержимым DX плюс значение идентификатора mas (не забывайте, что транслятор присваивает каждому идентификатору значение, равное смещению этого идентификатора относительно начала сегмента данных).

Косвенная индексная адресация со смещением

Косвенная индексная адресация со смещением очень похожа на косвенную базовую адресацию со смещением. Здесь также для формирования эффективного адреса используется один из регистров общего назначения. Но индексная адресация обладает одной интересной особенностью, которая очень удобна для работы с массивами. Она связана с возможностью так называемого *масштабирования* содержимого индексного регистра. Что это такое? Для выяснения смысла этого термина стоит вернуться к материалу главы 3. В контексте нашего обсуждения нас интересует байт sib. При описании структуры этого байта (см. раздел «Формат машинных команд IA-32» в главе 3) отмечалось, что он состоит из трех полей. Одно из этих полей — поле *масштаба* (ss), на значение которого умножается содержимое индексного регистра. К примеру, в команде mov ax,masfesi*2] значение эффективного адреса второго операнда определяется выражением mas+(esi)*2. В связи с тем, что в ассемблере нет средств индексации массивов, программисту приходится организовывать ее своими силами. Наличие возможности масштабирования существенно помогает в решении этой проблемы, но при условии, что размер элементов массива составляет 1, 2, 4 или 8 байт.

Косвенная базовая индексная адресация

При косвенной базовой индексной адресации эффективный адрес формируется как сумма содержимого двух регистров общего назначения: базового и индексного. В качестве этих регистров могут применяться любые регистры общего назначения, при этом часто содержимое индексного регистра масштабируется. Например:

```
mov  eax,[esi][edx]
```

В данном примере эффективный адрес второго операнда формируется из двух компонентов, (ESI) + (EDX).

Косвенная базовая индексная адресация со смещением

Косвенная базовая индексная адресация со смещением является дополнением косвенной индексной адресации. Эффективный адрес формируется как сумма трех составляющих: содержимого базового регистра, содержимого индексного регистра и значения поля смещения в команде. К примеру, команда mov eax,[esi+5][edx] пересылает в регистр EAX двойное слово по адресу: (ESI) + 5 + (EDX). Команда add ax,array[esi][ebx] складывает содержимое регистра AX с содержимым слова по адресу, образованному значением идентификатора array + (ESI) + (EBX).

Далее вернемся к обсуждению операндов команды и выясним, какие возможности предоставляются ассемблером для формирования более сложных операндов — операндов-выражений.

Операнды-выражения

Операнд команды может быть *выражением,* представляющим собой комбинацию операндов и *операторов* ассемблера. Транслятор ассемблера рассматривает выражение как единое целое и преобразует его в числовую константу. Логически значением этой константы может быть адрес некоторой ячейки памяти или некоторое абсолютное значение.

Перечислим возможные типы операторов ассемблера (TASM) и синтаксические правила формирования выражений ассемблера. Как и в языках высокого уровня, выполнение операторов ассемблера при вычислении выражений осуществляется в соответствии с их приоритетами (табл. 5.1). Операторы с одинаковыми приоритетами обрабатываются последовательно слева направо. Изменение порядка выполнения возможно путем расстановки круглых скобок, которые имеют наивысший приоритет.

Таблица 5.1. Операторы и их приоритет

Оператор	Приоритет
LENGTH, SIZE, WIDTH, MASK, (,), [,], <, >	1
.	2
:	3
PTR, OFFSET, SEG, TYPE, THIS	4
HIGH, LOW	5
+, - (унарные)	6
*, /, MOD, SHL, SHR	7

продолжение ⇨

Таблица 5.1 *(продолжение)*

Оператор	Приоритет
+, - (бинарные)	8
EQ, NE, LT, LE, GT, GE	9
NOT	10
AND	11
OR, XOR	12
SHORT, TYPE	13

Дадим краткую характеристику операторов. В приложении Б (http://www.piter.com/download) для сравнения и информации приведены сведения об операторах и предопределенных символах транслятора MASM.

9 *Арифметические операторы* (рис. 5.5). К арифметическим операторам относятся унарные и бинарные операторы «плюс» (+) и «минус» (-), а также операторы умножения (*), целочисленного деления (/), получения остатка от деления (MOD). Эти операторы в табл. 5.1 соответствуют уровням приоритета 6, 7, 8. Например,

```
tab_sizeequ 50          ; размер массива в байтах
size_el    equ 2           ; размер элементов
...
; вычисляется число элементов массива и заносится в регистр cx
mov cx,tab_size / size_el    ; оператор "/"
```

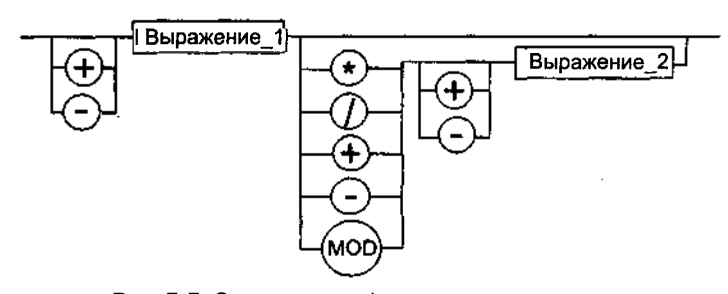


Рис. 5.5. Синтаксис арифметических операторов

Операторы сдвига выполняют сдвиг выражения на указанное количество разрядов (рис. 5.6). Например,

```
mask_b  equ 10111011
...
mov al,mask_b shr 3  ;al=00010111
```

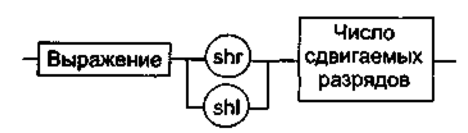


Рис. 5.6. Синтаксис операторов сдвига

- *Операторы сравнения* возвращают значение «истина» или «ложь» и предназначены для формирования логических выражений (рис. 5.7 и табл. 5.2). Логическое значение «истина» соответствует логической единице, а «ложь» — логическому нулю. Логическая единица — значение, все биты которого равны 1, соответственно, логический нуль — значение, все биты которого равны 0. Например,

```
tab_sizeequ 30       ; размер таблицы
...
mov al,tab_size ge 50   ; загрузка размера таблицы в al
cmp al,0             ; если tab_size < 50. то
je  ml                   ; переход на ml

mi: ...
```

В этом примере, если значение tab_size больше или равно 50, то результат в AL равен 0ffh, а если tab_size меньше 50, то результат в AL равен 00h. Команда CMP сравнивает значение AL с нулем и устанавливает соответствующие флаги в FLAGS/EFLAGS. Команда JE на основе анализа этих флагов передает или не передает управление на метку ml.

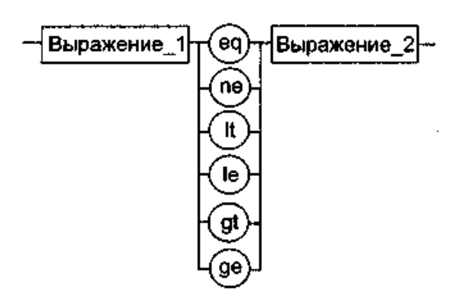


Рис. 5.7. Синтаксис операторов сравнения

Таблица 5.2. Операторы сравнения

Оператор	Значение
Eq	Истина, если выражение_1 равно выражение_2
Ne	Истина, если выражение_1 не равно выражение_2
Lt	Истина, если выражение 1 меньше выражение_2
Le	Истина, если выражение 1 меньше или равно выражение_2
Gt	Истина, если выражение 1 больше выражение 2
Ge	Истина, если выражение_1 больше или равно выражение_2

- *Логические операторы* выполняют над выражениями побитовые операции (рис. 5.8). Выражения должны быть абсолютными, то есть такими, численное значение которых может быть вычислено транслятором. Например,

```
flags   equ 10010011
mov al,flags xor 01h; al = 10010010; пересылка в al поля flags
                                ; с инвертированным правым битом
```

Более подробные сведения о правилах, в соответствии с которыми вычисляется результат логических операций, приводятся в главе 9.

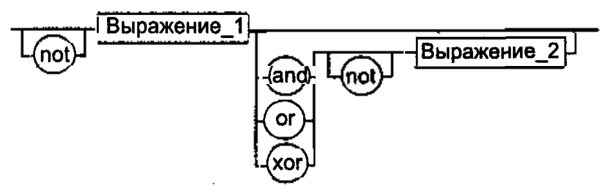


Рис. 5.8. Синтаксис логических операторов

- *Индексный оператор.* Как показано на рис. 5.9, квадратные скобки транслятор воспринимает как указание сложить значение **выражение_1** за этими скобками со значением **выражение_2**, заключенным в скобки. Например,

```
mov ax,mas[si]  ; пересылка слова по адресу mas + (si) в регистр ax
```

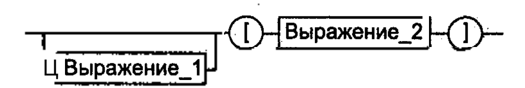


Рис. 5.9. Синтаксис индексного оператора

ПРИМЕЧАНИЕ В литературе принято следующее соглашение: когда в тексте речь идет о содержимом регистра, то его название берут в круглые скобки. К примеру, в нашем случае запись в тексте комментария последнего фрагмента программы mas + (si) означает вычисление выражения, составляющего значение смещения символического имени mas плюс содержимое регистра SI.

- *Оператор переопределения типа* PTR применяется для переопределения или уточнения типа метки (или переменной), определяемой выражением (рис. 5.10). Тип может принимать одно из следующих значений: BYTE, WORD, DWORD, QWORD, TBYTE, NEAR, FAR (что означают эти значения, вы узнаете далее в этой главе). Например,

```
d_wrd   dd  0
...
mov al,byte ptr d_wrd+1  ; пересылка второго байта из двойного слова
```

Поясним этот фрагмент программы. Переменная d_wrd имеет тип двойного слова. Что делать, если возникнет необходимость обращения не ко всему значению переменной, а только к одному из входящих в нее байтов (например, ко второму)? Если попытаться сделать это командой mov al,d_wrd+1, то транслятор выдаст сообщение о несовпадении типов операндов. Оператор PTR позволяет непосредственно в команде переопределить тип и выполнить команду.

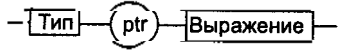


Рис. **5.10.** Синтаксис оператора переопределения типа

- *Оператор переопределения сегмента* заставляет вычислять физический адрес относительно конкретно задаваемой сегментной составляющей: «имя сегментного регистра», «имя сегмента» из соответствующей директивы SEGMENT или «имя группы» (рис. 5.11).

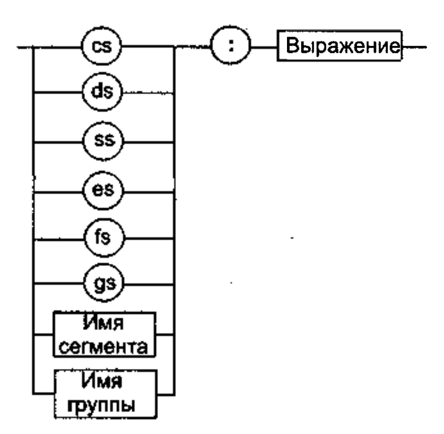


Рис. 5.11. Синтаксис оператора переопределения сегмента

Этот момент важен, поэтому поясним его подробнее. При обсуждении сегментации (см. главу 2) упоминалось о том, что процессор на аппаратном уровне поддерживает три типа сегментов — кода, стека и данных. В чем заключается такая аппаратная поддержка? К примеру, для выборки на выполнение очередной команды процессор должен обязательно посмотреть содержимое сегментного регистра CS, и только его. В этом регистре содержится информация о начале сегмента команд. В реальном режиме работы процессора в сегментном регистре CS находится не сдвинутый на четыре бита влево физический адрес начала сегмента. Для получения адреса конкретной команды процессору остается умножить содержимое CS на 16 (что означает сдвиг на четыре разряда) и сложить полученное 20-разрядное значение с 16-разрядным содержимым регистра IP. Примерно то же самое происходит и тогда, когда процессор обрабатывает операнды в машинной команде. Если он видит, что операнд — это адрес *(эффективный* адрес, который является только частью физического адреса), то он знает, в каком сегменте его искать, — по умолчанию это сегмент, адрес начала которого записан в сегментном регистре DS. В защищенном режиме сегментные регистры содержат селекторы, с помощью которых также можно получить доступ к информации об адресе начала сегмента.

А что же с сегментом стека? Для большей ясности стоит вернуться к вопросу о регистрах общего назначения (см. главу 2). В контексте нашего рассмотрения интерес представляют регистры SP и BP. Если процессор видит в качестве операнда (или его части, если операнд — выражение) один из этих регистров, то по умолчанию он формирует физический адрес операнда, используя содержимое регистра SS как сегментную составляющую этого адреса. Что подразумевает термин «по умолчанию»? Вспомним (см. главу 1) набор микропрограмм в блоке микропрограммного управления, каждая из которых выполняет одну из команд в системе машинных команд процессора. Каждая микропрограмма работает по своему алгоритму. Изменить его, конечно же, нельзя, но можно чуть-чуть подкорректировать. Делается это с помощью необязательного поля префикса машинной команды (см. раздел «Формат машинных команд IA-32» в главе 3). Если программист согласен с тем, как работает команда, то это поле отсутствует. Если же есть необходимость внести поправку (если, конечно, она допустима для конкретной команды) в алго-

ритм работы команды, то необходимо сформировать соответствующий префикс. *Префикс* представляет собой однобайтовую величину, численное значение которой определяет ее назначение. Процессор распознает по указанному значению, что этот байт является префиксом, и дальнейшая работа микропрограммы выполняется с учетом поступившего указания на корректировку ее работы. В главе 3 мы перечислили все возможные префиксы машинных команд. В контексте нашего обсуждения интерес представляет один из них — *префикс замены сегмента.* Его назначение состоит в том, чтобы указать процессору (а по сути, микропрограмме) на то, что мы не хотим использовать сегмент по умолчанию. Возможности для подобного переопределения, конечно, ограничены. Сегмент команд переопределить нельзя, адрес очередной исполняемой команды однозначно определяется парой CS:IP. А вот сегменты стека и данных — можно. Для этого и предназначен оператор переопределения сегмента (:). Транслятор ассемблера, обрабатывая этот оператор, формирует соответствующий однобайтовый префикс замены сегмента и ставит его перед машинным представлением соответствующей команды ассемблера. Например,

```
.code
...
    jmp met1     ; обход обязателен, иначе поле ind
                     ; будет трактоваться как очередная команда
ind      db   5      ; описание поля данных в сегменте команд
metl:
...
mov      al,cs:ind   ; переопределение сегмента позволяет работать с данными,
                     ; определенными внутри сегмента кода
```

Продолжим перечисление операторов.

- *Оператор именования типа структуры* также заставляет транслятор производить определенные вычисления, если он встречается в выражении. Подробно этот оператор (.) описывается в главе 13 при обсуждении сложных типов данных.
- *Оператор получения сегментной составляющей адреса выражения* возвращает физический адрес сегмента для выражения, в качестве которого могут выступать метка, переменная, имя сегмента, имя группы или некоторое символическое имя (рис. 5.12).



Рис. 5.12. Синтаксис оператора получения сегментной составляющей

- *Оператор получения смещения выражения* позволяет получить значение смещения выражения в байтах относительно начала того сегмента, в котором выражение определено (рис. 5.13).



Рис. 5.13. Синтаксис оператора получения смещения

Например,

```
.data
pole     dw  5
...
```

```
.code
...
mov ax,seg pole
mov es,ax
mov dx,offset pole  ; теперь в паре es:dx полный адрес pole
```

Директивы сегментации

В ходе предыдущего обсуждения были приведены основные правила записи команд и операндов в программе на ассемблере. Открытым остался вопрос о том, как правильно оформить последовательность команд, чтобы транслятор мог их обработать, а процессор — выполнить. В главах 2 и 3 мы уже касались понятия сегмента. При рассмотрении архитектуры процессора мы узнали, что он имеет шесть сегментных регистров, посредством которых может одновременно работать:

- с одним сегментом кода;
- с одним сегментом стека;
- с одним сегментом данных;
- с тремя дополнительными сегментами данных.

Еще раз вспомним, что физически сегмент представляет собой область памяти, занятую командами и/или данными, адреса которых вычисляются относительно значения в соответствующем сегментном регистре.

Синтаксическое описание сегмента на ассемблере представляет собой конструкцию, представленную на рис. 5.14.

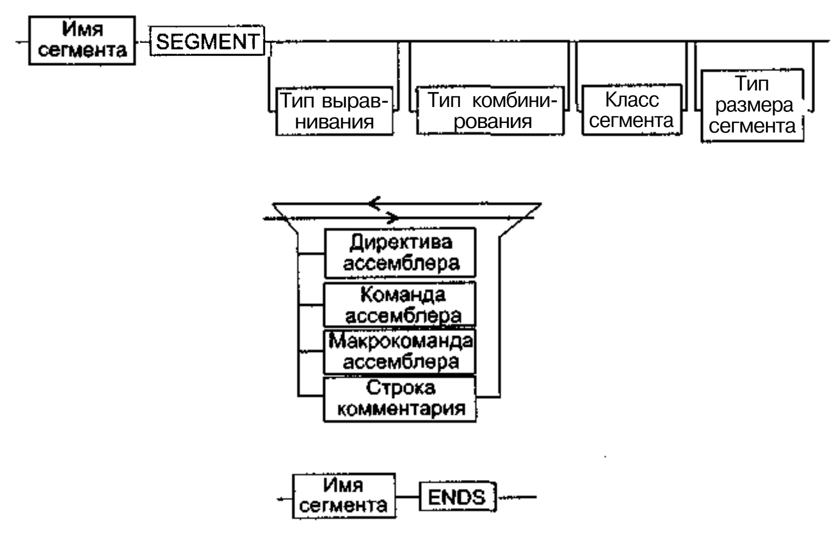


Рис. 5.14. Синтаксис описания сегмента

Важно отметить, что функциональное назначение сегментации несколько шире, чем простое разбиение программы на блоки кода, данных и стека. Сегментация является частью более общего механизма, связанного с концепцией модульного программирования. Она предполагает унификацию формата объектных модулей,

создаваемых компилятором, в том числе компилируемых с разных языков программирования. Это позволяет объединять программы, написанные на разных языках. Именно для реализации различных вариантов такого объединения и предназначены операнды в директиве SEGMENT. Рассмотрим их подробнее.

- *Атрибут выравнивания сегмента* (тип выравнивания) сообщает компоновщику о том, что нужно обеспечить размещение начала сегмента на заданной границе. Это важно, поскольку при правильном выравнивании доступ к данным в процессорах i80x86 выполняется быстрее. Допустимые значения этого атрибута приведены далее. По умолчанию тип выравнивания имеет значение PARA:
  - BYTE — выравнивание не выполняется. Сегмент может начинаться с любого адреса памяти;
  - WORD — сегмент начинается по адресу, кратному двум, то есть последний (младший) значащий бит физического адреса равен 0 (выравнивание по границе слова);
  - DWORD — сегмент начинается по адресу, кратному четырем, то есть два последних (младших) значащих бита равны 0 (выравнивание по границе двойного слова);
  - PARA — сегмент начинается по адресу, кратному 16, то есть последняя шестнадцатеричная цифра адреса должна быть 0h (выравнивание по границе параграфа);
  - PAGE — сегмент начинается по адресу, кратному 256, то есть две последние шестнадцатеричные цифры должны быть 00h (выравнивание по границе страницы размером 256 байт);
  - MEMPAGE — сегмент начинается по адресу, кратному 4 Кбайт, то есть три последние шестнадцатеричные цифры должны быть 000h (адрес следующей страницы памяти размером 4 Кбайт).
- *Атрибут комбинирования сегментов* (комбинаторный тип) сообщает компоновщику, как нужно комбинировать сегменты различных модулей, имеющие одно и то же имя. По умолчанию атрибут комбинирования принимает значение PRIVATE. Возможные значения атрибута комбинирования сегмента перечислены далее:
  - PRIVATE — сегмент не будет объединяться с другими сегментами с тем же именем вне данного модуля;
  - PUBLIC — заставляет компоновщик объединить все сегменты с одинаковым именем. Новый объединенный сегмент будет целым и непрерывным. Все адреса (смещения) объектов, а это могут быть, в зависимости от типа сегмента, команды или данные, будут вычисляться относительно начала этого нового сегмента;
  - COMMON — располагает все сегменты с одним и тем же именем по одному адресу, то есть все сегменты с данным именем перекрываются. Размер полученного в результате сегмента будет равен размеру самого большого сегмента;
  - AT xxxx — располагает сегмент по абсолютному адресу параграфа *(параграф* — область памяти, объем которой кратен 16, потому последняя шестнадцате-

ричная цифра адреса параграфа равна 0). Абсолютный адрес параграфа задается выражением ххх. Компоновщик располагает сегмент по заданному адресу памяти (это можно использовать, например, для доступа к видеопамяти или области ПЗУ), учитывая атрибут комбинирования. Физически это означает, что сегмент при загрузке в память будет расположен начиная с этого абсолютного адреса параграфа, но для доступа к нему в соответствующий сегментный регистр должно быть загружено заданное в атрибуте значение. Все метки и адреса в определенном таким образом сегменте отсчитываются относительно заданного абсолютного адреса;

- STACK — определение *сегмента стека.* Заставляет компоновщик объединить все одноименные сегменты и вычислять адреса в этих сегментах относительно регистра SS. Комбинированный тип STACK (стек) аналогичен комбинированному типу PUBLIC за исключением того, что регистр SS является стандартным сегментным регистром для сегментов стека. Регистр SP устанавливается на конец объединенного сегмента стека. Если не указано ни одного сегмента стека, компоновщик выдаст предупреждение, что стековый сегмент не найден. Если сегмент стека создан, а комбинированный тип STACK не используется, программист должен явно загрузить в регистр SS адрес сегмента (подобно тому, как это делается для регистра DS).

- *Атрибут класса сегмента* (тип класса) — это заключенная в кавычки строка, помогающая компоновщику определить нужный порядок следования сегментов при сборке программы из сегментов нескольких модулей. Компоновщик объединяет вместе в памяти все сегменты с одним и тем же именем класса (имя класса в общем случае может быть любым, но лучше, если оно отражает функциональное назначение сегмента). Типичным примером использования имени класса (обычно класса code) является объединение в группу всех сегментов кода программы. С помощью механизма типизации класса можно группировать также сегменты инициализированных и неинициализированных данных.
- *Атрибут размера сегмента.* Для процессоров i80386 и выше сегменты могут быть 16- или 32-разрядными. Это влияет прежде всего на размер сегмента и порядок формирования физического адреса внутри него. Далее перечислены возможные значения атрибута:
  - USE16 — сегмент допускает 16-разрядную адресацию. При формировании физического адреса может использоваться только 16-разрядное смещение. Соответственно, такой сегмент может содержать до 64 Кбайт кода или данных;
  - USE32 — сегмент должен быть 32-разрядным. При формировании физического адреса может использоваться 32-разрядное смещение. Поэтому такой сегмент может содержать до 4 Гбайт кода или данных.

Все сегменты сами по себе равноправны, так как директивы SEGMENT и ENDS не содержат информации о функциональном назначении сегментов. Для того чтобы использовать их как сегменты кода, данных или стека, необходимо предварительно сообщить транслятору об этом с помощью специальной директивы ASSUME, формат которой показан на рис. 5.15. Эта директива сообщает транслятору, какой сег-

мент к какому сегментному регистру привязан. В свою очередь, это позволяет транслятору корректно связывать символические имена, определенные в сегментах. Привязка сегментов к сегментным регистрам осуществляется с помощью операндов этой директивы, в которых имя_сегмента должно быть именем сегмента, определенным в исходном тексте программы директивой SEGMENT или ключевым словом NOTHING. Если в качестве операнда используется только ключевое слово NOTHING, то предшествующие назначения сегментных регистров аннулируются, причем сразу для всех шести сегментных регистров. Ключевое слово NOTHING можно также использовать вместо аргумента имя сегмента; в этом случае будет выборочно разрываться связь между сегментом с именем имя сегмента и соответствующим сегментным регистром.

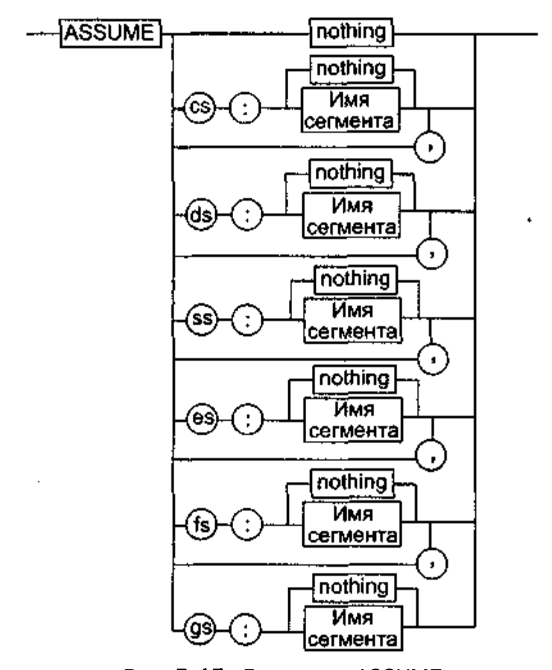


Рис. 5.15. Директива ASSUME

Рассмотренные ранее директивы сегментации используются для оформления программы в трансляторах MASM и TASM. Поэтому их называют *стандартными* директивами сегментации.

Для простых программ, содержащих по одному сегменту для кода, данных и стека, хотелось бы упростить их описание. Для этого в трансляторы MASM и TASM была введена возможность использования *упрощенных* директив сегментации. При этом возникла проблема, связанная с тем, что необходимо было как-то компенсировать невозможность напрямую управлять размещением и комбинированием сегментов. Для этого совместно с упрощенными директивами сегментации стали использовать директиву указания модели памяти MODEL, которая частично стала управлять размещением сегментов и выполнять функции директивы ASSUME (поэтому при введении в программу упрощенных директив сегментации директиву

ASSUME можно не указывать). Директива MODEL связывает сегменты, которые при наличии упрощенных директив сегментации имеют предопределенные имена, с сегментными регистрами (хотя все равно придется явно инициализировать регистр DS).

Для сравнения приведем два листинга с программами на ассемблере. Функционально они одинаковы и выводят на консоль сообщение: «Hello World! No war and bomb! Let's live friendly and learn assembler language.». Листинг 5.1 содержит программу со стандартными директивами сегментации, а листинг 5.2, соответственно, — с упрощенными.

Листинг 5.1. Использование стандартных директив сегментации

```
data    segment para public 'data'
message db  'Hello World! No war and bomb! Let us live friendly and learn
assembler language. $'
data    ends
stk segment stack
    db  256 dup ('?')                 ;сегмент стека
stk ends
code    segment para public 'code'  ; начало сегмента кода
main    proc                        ; начало процедуры main
    assume cs:code,ds:data,ss:stk
    mov ax,data                     ; адрес сегмента данных в регистр ах
    mov ds,ax                       ; ах в ds
    mov ah,9
    mov dx,offset message
    int 21h                         ; вывод сообщения на экран
    mov ax,4c00h                    ; пересылка 4c00h в регистр ах
    int 21h                         ; вызов прерывания с номером 21h
main    endp                        ; конец процедуры main
code    ends                        ; конец сегмента кода
end main                            ; конец программы с точкой входа main
```

Листинг 5.2. Использование упрощенных директив сегментации

```
masm                ; режим работы для TASM - masm, для MASM - не нужно
model   small       ; модель памяти
.data               ; сегмент данных
message db  'Hello World! No war and bomb! Let us live friendly and learn
assembler language. $'
.stack  256h        ; сегмент стека
.code               ; сегмент кода
main    proc        ; начало процедуры main
    mov ax,@data    ; заносим адрес сегмента данных в регистр ах
    mov ds,ax       ; ах в ds
    mov ah,9
    mov dx,offset message
    int 21h         ; вывод сообщения на экран
    mov ax,4c00h    ; пересылка 4c00h в регистр ах
    int 21h         ; вызов прерывания с номером 21h
main    endp        ; конец процедуры main
end main            ; конец программы с точкой входа main
```

Синтаксис директивы MODEL показан на рис. 5.16.

Обязательным параметром директивы MODEL является **модель_памяти**. Этот параметр определяет модель сегментации памяти для программного модуля. Предполагается, что программный модуль может иметь только определенные типы сегментов, которые определяются так называемыми упрощенными директивами описания сегментов (табл. 5.3).

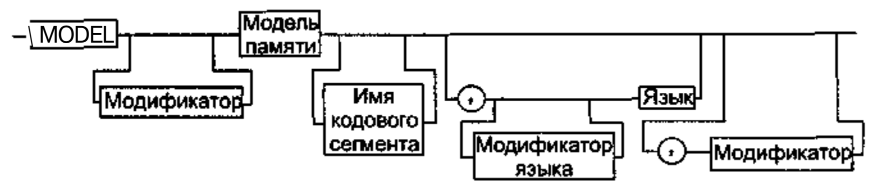


Рис. 5.16. Синтаксис директивы MODEL

Таблица 5.3. Упрощенные директивы определения сегмента

Формат директивы (режим **MASM)**	Назначение
.CODE [имя]	Начало или продолжение сегмента кода
.DATA	Начало или продолжение сегмента инициализированных данных. Также используется для определения данных типа near 1
.CONST	Начало или продолжение сегмента постоянных данных (констант) модуля
.DATA?	Начало или продолжение сегмента неинициализированных данных. Также используется для определения данных типа near
STACK [размер]	Начало или продолжение сегмента стека модуля. Параметр [размер] задает размер стека
.FARDATA [имя]	Начало или продолжение сегмента инициализированных данных тина far
.FARDATA? [имя]	Начало или продолжение сегмента неинициализированных данных типа far

Наличие в некоторых директивах параметра [имя] говорит о том, что возможно определение нескольких сегментов этого типа. С другой стороны, наличие нескольких видов сегментов данных обусловлено требованием обеспечить совместимость с некоторыми компиляторами языков высокого уровня, которые создают разные сегменты данных для инициализированных и неинициализированных данных, а также констант.

При использовании директивы MODEL транслятор делает доступными несколько предопределенных идентификаторов, к которым можно обращаться во время работы программы в целях получения информации о тех или иных характеристиках выбранной модели памяти (см. ниже). Перечислим эти идентификаторы и их значения для транслятора TASM (табл. 5.4). Предопределенные идентификаторы MASM приведены в приложении Б (см. на сайте http://www.piter.com/download).

Таблица 5.4. Идентификаторы, создаваемые директивой MODEL

Имя идентификатора	Значение переменной
@code	Физический адрес сегмента кода
@data	Физический адрес сегмента данных типа near
@fardata	Физический адрес сегмента данных типа far

Имя идентификатора	Значение переменной
@fardata?	Физический адрес сегмента неинициализированных данных типа far
@curseg	Физический адрес сегмента инициализированных данных типа far
@stack	Физический адрес сегмента стека

Если вернуться к листингу 5.2, то можно увидеть пример использования одного из перечисленных идентификаторов, а именно @data. Цель его применения — получение значения физического адреса сегмента данных программы.

В заключение обсуждения директивы MODEL отметим, что операнды директивы MODEL задают модель памяти, определяющей набор сегментов программы, размеры сегментов данных и кода, способы связывания сегментов и сегментных регистров. В табл. 5.5 приведены некоторые значения параметров модели памяти директивы MODEL

Таблица 5.5. Модели памяти

Модель	Тип кода	Тип данных	Назначение модели
TINY	near	near	Код и данные объединены в одну группу с именем DGROUP. Используется для создания программ файлового формата COM
SMALL	near	near	Код занимает один сегмент, данные объединены в одну группу с именем DGROUP. Эту модель обычно используют для большинства программ на ассемблере
MEDIUM	far	near	Код занимает несколько сегментов, по одному на каждый объединяемый программный модуль. Все ссылки на передачу управления имеют тип far. Данные объединены в одной группе; все ссылки на них имеют тип near
COMPACT	near	far	Код в одном сегменте; ссылки на данные имеют тип far
LARGE	far	far	Код в нескольких сегментах, по одному на каждый объединяемый программный модуль
FLAT	near	near	Код и данные в одном 32-битном сегменте (плоская модель памяти)

Параметр модификатор директивы MODEL позволяет уточнить некоторые особенности использования выбранной модели памяти (табл. 5.6).

Таблица 5.6. Модификаторы модели памяти

Значение модификатора	**Назначение**
use16	Сегменты выбранной модели используются как 16-разрядные (если соответствующей директивой указан процессор i80386 или i80486)

продолжение ⇨

Таблица 5.6 *(продолжение)*

Значение модификатора	Назначение
use32	Сегменты выбранной модели используются как 32-разрядные (если соответствующей директивой указан процессор i80386 или i80486)
dos	Программа будет работать в MS-DOS

Необязательные параметры язык и модификатор_языка определяют некоторые особенности вызова процедур. Необходимость в использовании этих параметров появляется при написании и связывании программ на разных языках программирования. Этого вопрос будет подробнее обсужден в главе 15 при рассмотрении средств модульного программирования на ассемблере.

Описанные ранее стандартные и упрощенные директивы сегментации не исключают друг друга. Стандартные директивы используются, когда программист желает получить полный контроль над размещением сегментов в памяти и их комбинированием с сегментами других модулей. Упрощенные директивы целесообразно использовать для простых программ, а также для программ, предназначенных для связывания с программными модулями, написанными на языках высокого уровня. Это позволяет компоновщику эффективно связывать модули разных языков за счет стандартизации связей и управления.

Таким образом, в ходе предыдущего изложения была описана структура программы на ассемблере. Было показано, что программа разбивается на несколько сегментов, каждый из которых имеет свое функциональное назначение. Для такого разбиения используются директивы сегментации. Трансляторы TASM и MASM предоставляют два типа таких директив: стандартные и упрощенные. Далее в учебнике будут использоваться упрощенные директивы сегментации, если не появится необходимость в применении стандартных директив.

Вторую часть главы мы посвятим вопросу описания данных в программе на ассемблере и организации доступа к ним.

Простые типы данных ассемблера

Любая программа предназначена для обработки некоторой информации, поэтому вопрос о том, какие типы данных языка программирования доступны для использования и какие средства языка привлекаются для их описания, обычно встает одним из первых. Трансляторы TASM и MASM предоставляют широкий набор средств описания и обработки данных, который вполне сравним с аналогичными средствами большинства языков высокого уровня.

Понятие *типа данных* носит двойственный характер. С точки зрения размерности процессор аппаратно поддерживает следующие основные типы данных (рис. 5.17).

- *Байт* — восемь последовательно расположенных битов, пронумерованных от 0 до 7, при этом бит 0 является самым младшим значащим битом.

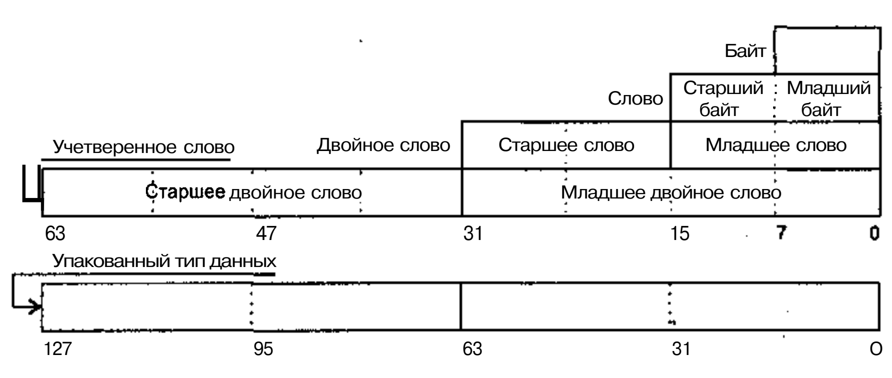


Рис. 5.17. Основные типы данных процессора

- *Слово* — последовательность из двух байтов, имеющих последовательные адреса. Размер слова — 16 битов; биты в слове нумеруются от 0 до 15. Байт, содержащий нулевой бит, называется *младшим байтом,* а байт, содержащий 15-й бит, — *старшим.* Процессоры Intel имеют важную особенность — младший байт всегда хранится по меньшему адресу. *Адресом слова* считается адрес его младшего байта. Адрес старшего байта может быть использован для доступа к старшей половине слова.
- *Двойное слово* — последовательность из четырех байтов (32 бита), расположенных по последовательным адресам. Нумерация этих битов производится от О до 31. Слово, содержащее нулевой бит, называется *младшим словом,* а слово, содержащее 31-й бит, — *старшим словом.* Младшее слово хранится по меньшему адресу. *Адресом двойного слова* считается адрес его младшего слова. Адрес старшего слова может быть использован для доступа к старшей половине двойного слова.
- *Учетверенное слово* — последовательность из восьми байтов (64 бита), расположенных по последовательным адресам. Нумерация битов производится от О до 63. Двойное слово, содержащее нулевой бит, называется *младшим двойным словом,* а двойное слово, содержащее 63-й бит, — *старшим двойным словом.* Младшее двойное слово хранится по меньшему адресу. *Адресом учетверенного слова* считается адрес его младшего двойного слова. Адрес старшего двойного слова может быть использован для доступа к старшей половине учетверенного слова.
- *128-битный упакованный тип данных.* Этот тип данных появился в процессоре Pentium III. Для работы с ним в процессор введены специальные команды.

Кроме трактовки типов данных с точки зрения их разрядности, процессор на уровне команд поддерживает *логическую* интерпретацию этих типов, как показано на рис. 5.18 (Зн означает знаковый бит).

- *Целый тип со знаком* — двоичное значение со знаком размером 8, 16 или 32 бита. Знак в этом двоичном числе содержится в 7, 15 или 31 бите соответственно. Ноль в этих битах в операндах соответствует положительному числу, а единица —

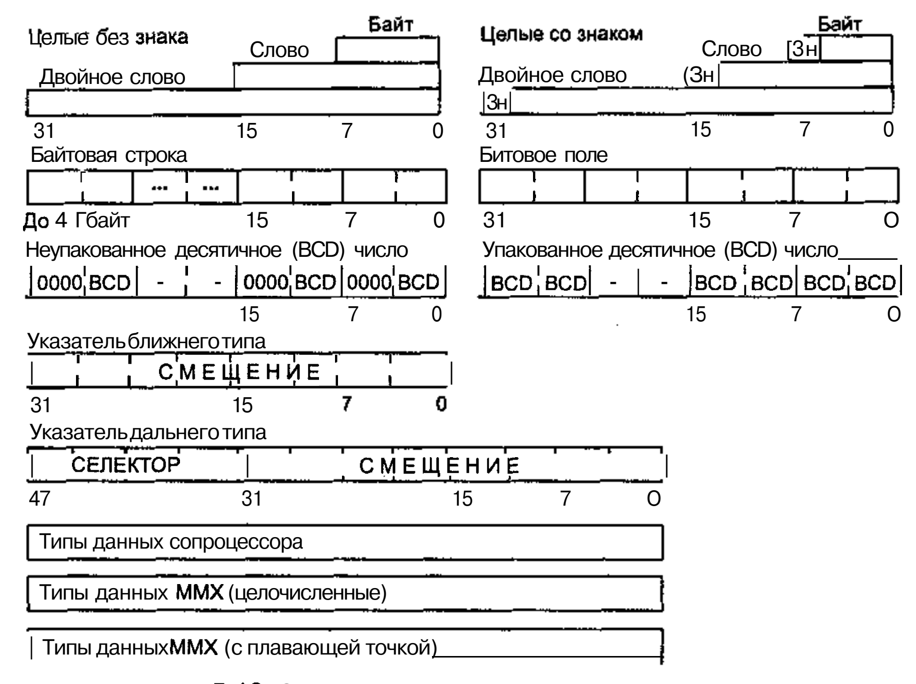


Рис. 5.18. Основные логические типы данных процессора

отрицательному. Отрицательные числа представляются в дополнительном коде. Числовые диапазоны для этого типа данных следующие:

- 8-разрядное целое — от –128 до +127;
- 16-разрядное целое — от -32 768 до +32 767;
- 32-разрядное целое — от -2^{31} до $+2^{31}$ - 1.

- *Целый тип без знака* — двоичное значение *без знака* размером 8, 16 или 32 бита. Числовой диапазон для этого типа следующий:
  - байт — от 0 до 255;
  - слово — от 0 до 65 535;
  - двойное слово— от 0 до 2^{32} - 1.
- *Указательна память* бывает двух типов:
  - *ближний тип* — 32-разрядный логический адрес, представляющий собой относительное смещение в байтах от начала сегмента; указатели подобного типа могут также использоваться в сплошной (плоской) модели памяти, где сегментные составляющие одинаковы;
  - *дальний тип* — 48-разрядный логический адрес, состоящий из двух частей: 16-разрядной сегментной части *(селектора)* и 32-разрядного смещения.
- *Цепочка* представляет собой некоторый непрерывный набор байтов, слов или двойных слов максимальной длиной до 4 Гбайт.

ж *Битовое поле* представляет собой непрерывную последовательность битов, в которой каждый бит является независимым и может рассматриваться как отдельная переменная. Битовое поле может начинаться с любого бита любого байта и содержать до 32 битов.

- *Неупакованный двоично-десятичный тип* — байтовое представление десятичной цифры от 0 до 9. Неупакованные десятичные числа хранятся как байтовые значения без знака по одной цифре в каждом байте. Значение цифры определяется младшим полубайтом.
- *Упакованный двоично-десятичный тип* представляет собой упакованное представление двух десятичных цифр от 0 до 9 в одном байте. Каждая цифра хранится в своем полубайте. Цифра в старшем полубайте (биты 4-7) является старшей.
- *Типы данных с плавающей точкой.* Сопроцессор имеет несколько собственных типов данных, несовместимых с типами данных целочисленного устройства. Подробно этот вопрос обсуждается в главе 17.

* *Типы данных MMX-расширения Pentium MMX/II/III/IV.* Данный тип данных появился в процессоре Pentium MMX. Он представляет собой совокупность упакованных целочисленных элементов определенного размера.

9 *Типы данных MMX-расширения Pentium III/IV.* Этот тип данных появился в процессоре Pentium III. Он представляет собой совокупность упакованных элементов с плавающей точкой фиксированного размера.

Описанные ранее данные можно определить как данные *простого* типа. Описать их можно с помощью специального вида директив — *директив резервирования и инициализации данных.* Эти директивы, по сути, являются указаниями транслятору на выделение определенного объема памяти. Если проводить аналогию с языками высокого уровня, то директивы резервирования и инициализации данных являются определениями переменных. Машинного эквивалента этим (впрочем, как и другим) директивам нет; просто транслятор, обрабатывая каждую такую директиву, выделяет необходимое количество байтов памяти и при необходимости инициализирует эту область некоторым значением. Формат директив резервирования и инициализации данных простых типов показан на рис. 5.19.

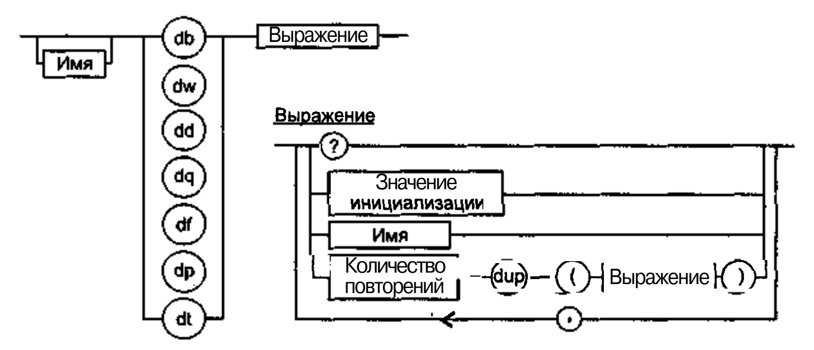


Рис. 5.19. Директивы описания данных простых типов

На рисунке использованы следующие обозначения.

* Знак вопроса (?) показывает, что содержимое поля не определено, то есть при задании директивы с таким значением выражения содержимое выделенного участка физической памяти изменяться не будет. Фактически, создается неинициализированная переменная.

Ж Значение инициализации — значение элемента данных, которое будет занесено в память после загрузки программы. Фактически, создается инициализированная переменная, в качестве которой могут выступать константы, строки символов, константные и адресные выражения в зависимости от типа данных.

Выражение — итеративная конструкция, о синтаксисе которой можно судить по рисунку. В частности, она позволяет повторить занесение в физическую память выражения в скобках столько раз, сколько повторений указано.

Имя — некоторое символическое имя метки или ячейки памяти в сегменте данных, используемое в программе.

Далее представлены поддерживаемые TASM и MASM директивы резервирования и инициализации данных, а также информация о возможных типах и диапазонах значений, которые можно описывать или задавать с их помощью.

я DB — резервирование памяти для данных размером 1 байт. Директивой DB можно задавать следующие значения:

D выражение или константу, принимающую значение из диапазона −128...+127 (для чисел со знаком) или 0...255 (для чисел без знака);

□ 8-разрядное относительное выражение, использующее операции HIGH и LOW;

D символьную строку из одного или более символов, которая заключается в кавычки (в этом случае определяется столько байтов, сколько символов в строке).

til DW — резервирование памяти для данных размером два байта. Директивой DW можно задавать следующие значения:

□ выражение или константу, принимающую значение из диапазона -32 768...32 767 (для чисел со знаком) или 0...65 535 (для чисел без знака);

D выражение, занимающее 16 или менее битов, в качестве которого может выступать смещение в 16-битовом сегменте или адрес сегмента;

D 1- или 2-байтовая строка, заключенная в кавычки.

DD — резервирование памяти для данных размером четыре байта. Директивой DD можно задавать следующие значения:

D выражение или константу, принимающую значение из диапазона -32 768...+32 767 (для чисел со знаком и процессора i8086), 0...65 535 (для чисел без знака и процессора i8086), -2 147 483 648...+2 147 483 647 (для чисел со знаком и процессора i386 и выше) или 0...4 294 967 295 (для чисел без знака и процессора i386 и выше);

□ относительное или адресное выражение, состоящее из 16-разрядного адреса сегмента и 16-разрядного смещения;

О строку длиной до 4 символов, заключенную в кавычки.

- DF и DP — резервирование памяти для данных размером 6 байтов. Директивами DF и DP можно задавать следующие значения:
  - выражение или константу, принимающую значение из диапазона -32 768...+32 767 (для чисел со знаком и процессора i8086), 0...65 535 (для чисел без знака и процессора i8086), -2 147 483 648...+2 147 483 647 (для чисел со знаком и процессора i386 и выше) или 0...4 294 967 295 (для чисел без знака и процессора i386 и выше);
  - относительное или адресное выражение, состоящее из 32 или менее битов (для i80386) или 16 или менее битов (для первых моделей процессоров Intel);
  - адресное выражение, состоящее из 16-разрядного сегмента и 32-разрядного смещения;
  - константу со знаком из диапазона $-2^{47}...2^{47} - 1$;
  - константу без знака из диапазона $0...2^{48} - 1$;
  - строку длиной до 6 байтов, заключенную в кавычки.
- DQ — резервирование памяти для данных размером 8 байтов. Директивой DQ можно задавать следующие значения:
  - выражение или константу, принимающую значение из диапазона -32 768...+32 767 (для чисел со знаком и процессора i8086), 0...65 535 (для чисел без знака и процессора i8086), -2 147 483 648...+2 147 483 647 (для чисел со знаком и процессора i386 и выше) или 0...4 294 967 295 (для чисел без знака и процессора i386 и выше);
  - относительное или адресное выражение, состоящее из 32 или менее битов (для i80386) или 16 или менее битов (для первых моделей процессоров Intel);
  - константу со знаком из диапазона $-2^{63}...2^{63} - 1$;
  - константу без знака из диапазона $0...2^{64} - 1$;
  - строку длиной до 8 байтов, заключенную в кавычки.
- DT — резервирование памяти для данных размером 10 байтов. Директивой DT можно задавать следующие значения:
  - выражение или константу, принимающую значение из диапазона -32 768...+32 767 (для чисел со знаком и процессора i8086), 0...65 535 (для чисел без знака и процессора i8086), -2 147 483 648...+2 147 483 647 (для чисел со знаком и процессора i386 и выше) или 0...4 294 967 295 (для чисел без знака и процессора i386 и выше);
  - относительное или адресное выражение, состоящее из 32 или менее битов (для i80386) или 16 или менее битов (для первых моделей);
  - адресное выражение, состоящее из 16-разрядного сегмента и 32-разрядного смещения;
  - константу со знаком из диапазона $-2^{79}...2^{79} - 1$;
  - константу без знака из диапазона $0...2^{80} - 1$;
  - строку длиной до 10 байтов, заключенную в кавычки;
  - упакованную десятичную константу в диапазоне 0...99 999 999 999 999 999 999.

Заметим, что все директивы позволяют задавать строковые значения, но нужно помнить, что в памяти эти значения могут выглядеть совсем не так, как они были описаны в директиве. Причиной этому является упоминавшийся ранее принцип «младший байт по младшему адресу». Для определения строк лучше использовать директиву DB. Задаваемые таким образом строки должны заключаться в кавычки. Эти кавычки могут быть одинарными (' ') или двойными (" "). Если задать в строке подряд два таких ограничителя, то вторая кавычка (одинарная или двойная) будет частью строки.

Для иллюстрации принципа «младший байт по младшему адресу» рассмотрим листинг 5.3, в котором определим сегмент данных. В этом сегменте данных приведено несколько директив описания простых типов данных.

Листинг 5.3. Пример использования директив резервирования и инициализации данных

```
masm
model small
.stack 100h
.data
message db "Запустите эту программу в отладчике",'$'
perem_1 db  0ffh
perem_2 dw  3a7fh
perem_3 dd  0f54d567ah
mas     db  10 dup (" ")
pole_l db 5 dup (?)
adr     dw  perem_3
adr_full    dd  perem_3
fin     db  "Конец сегмента данных программы $"
.code
start:
    mov ax,@data
    mov ds,ax
    mov ah,09h
    mov dx,offset message
    int     21h
    mov ax,4c00h
    int 21h
end     start
```

ВНИМАНИЕ Если при дальнейшем изложении вам что-то будет непонятно, можно порекомендовать на время прерваться и вновь вернуться к этому материалу после изучения главы 6.

Итак, наша цель — посмотреть, как выглядит в памяти компьютера сегмент данных программы, представленной в листинге 5.3. Это даст нам возможность обсудить практическую реализацию обозначенного нами принципа размещения данных.

1. Запустите отладчик td.exe, входящий в комплект поставки транслятора TASM.
2. Введите код, представленный в листинге 5.3, и сохраните его в виде файла с названием prg_5_3.asm. Все манипуляции с файлом будем производить в каталоге work, где должны содержаться все необходимые для компиляции, компоновки и отладки файлы пакета TASM.
3. Запустите процесс трансляции файла следующей командой:

```
tasm.exe /zi prg_5_2.asm , , ,
```

4. После устранения синтаксических ошибок запустите процесс компоновки объектного файла:

   ```
   tlink.exe /v prg_5_2.obj
   ```

5. Теперь можно производить отладку:

   ```
   td prg_5_2.exe
   ```

Если все было сделано правильно, то в отладчике откроется окно Module с исходным текстом программы. Для того чтобы с помощью отладчика просмотреть область памяти, содержащую наш сегмент данных, необходимо открыть окно Dump. Это делается с помощью команды View ▸ Dump главного меню.

Но одного открытия окна недостаточно, нужно еще настроить его на адрес начала сегмента данных. Этот адрес должен содержаться в сегментном регистре DS, но перед началом выполнения программы адрес в DS не соответствует началу сегмента данных. Нужно перед первым обращением к любому символическому имени произвести загрузку действительного физического адреса сегмента данных. Обычно это действие не откладывают и производят первыми двумя командами в сегменте кода. Действительный физический адрес сегмента данных извлекают как значение предопределенной переменной @data. В нашей программе эти действия выполняют команды

```
mov ax,@data
mov ds,ax
```

Для того чтобы посмотреть содержимое нашего сегмента данных, нужно остановить выполнение программы после этих двух команд. Это можно сделать, если перевести отладчик в пошаговый режим с помощью клавиши F7 или F8. Нажмите два раза клавишу F8. Теперь можно открыть окно Dump.

В окне Dump вызовите контекстное меню, щелкнув правой кнопкой мыши, и выберите команду Goto. Появится диалоговое окно, в котором нужно ввести начальный адрес памяти, начиная с которого информация будет выводиться в окне Dump. Синтаксис задания этого адреса должен соответствовать синтаксису задания адресного операнда в программе на ассемблере. Если нужно увидеть содержимое памяти для сегмента данных, начиная с начала, введите ds:0000 (рис. 5.20). Для удобства, если сегмент довольно велик, это окно можно развернуть на весь экран. Для этого нужно щелкнуть на значке в виде стрелки (↑) в правом верхнем углу окна Dump.

На рисунке представлено содержание сегмента данных программы из листинга 5.3 в двух представлениях: шестнадцатеричном и символьном. Видно, что со смещением 0000 расположены символы, входящие в строку message. Она занимает 34 байта. После нее следует байт, имеющий в сегменте данных символическое имя perem_1, содержимое этого байта — offh. Теперь обратите внимание на то, как размещены в памяти байты, входящие в слово, обозначенное символическим именем perem_2. Сначала следует байт со значением 7fh, а затем — со значением 3ah. Как видим, в памяти действительно сначала расположен младший байт значения, а затем старший. Теперь проанализируйте порядок размещения байтов для поля, обозначенного символическим именем perem_3. Оставшаяся часть сегмента данных не представляет трудности для самостоятельного анализа. Есть смысл остановиться лишь на двух специфических особенностях использования директив резервиро-

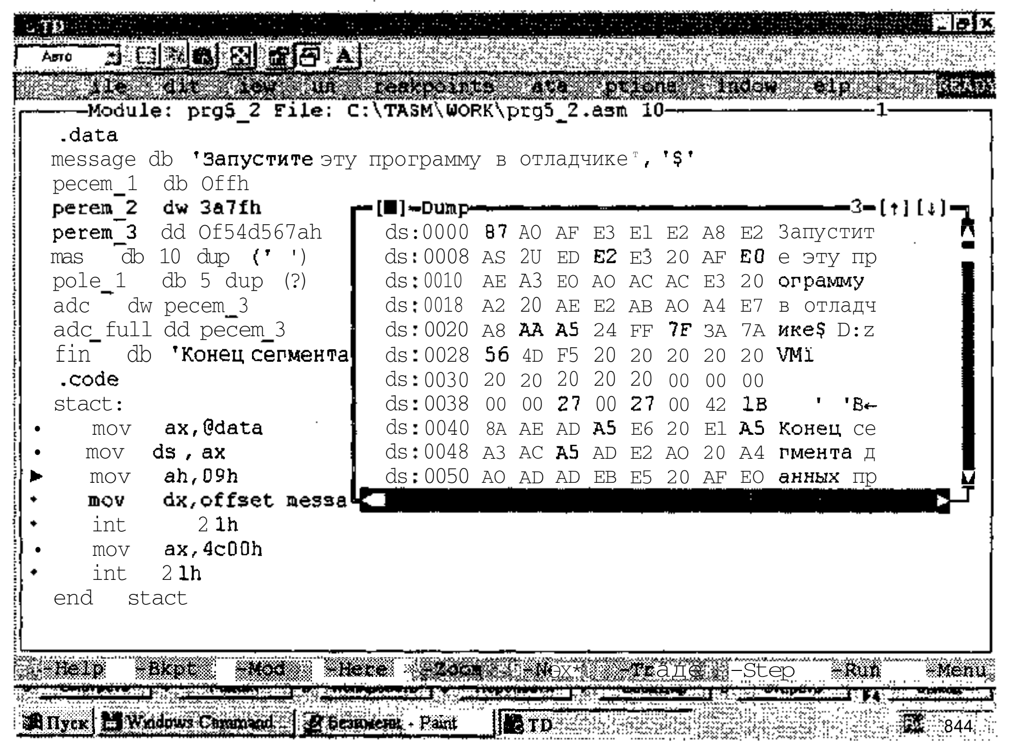


Рис. 5.20. Окно дампа памяти

вания и инициализации памяти. Речь идет об указании в поле операндов директив DW и DD идентификатора из поля имени этой или другой директивы резервирования и инициализации памяти. В нашем примере сегмента данных это директивы с идентификаторами adr и adr_full. Когда транслятор встречает директивы описания памяти с подобными операндами, то он формирует в памяти значения адресов тех переменных, чьи идентификаторы указаны в качестве операндов. В зависимости от директивы, применяемой для получения такого адреса, формируется либо полный адрес (директива DD) в виде двух байтов сегментного адреса и двух байтов смещения, либо только смещение (директива dw). Для тренировки найдите в представленном дампе поля, соответствующие идентификаторам adr и adr_full, и проанализируйте их содержимое.

Любой переменной, объявленной с помощью директив описания простых типов данных, ассемблер присваивает три атрибута:

- сегмент (seg) — адрес начала сегмента, содержащего переменную;
- смещение (offset) в байтах от начала сегмента с переменной;
- тип (type) — объем памяти, выделяемой переменной в соответствии с директивой объявления переменной.

Получить и использовать значение этих атрибутов в программе можно с помощью операторов ассемблера SEG, OFFSET и TYPE.

В заключение отметим, что в языке ассемблера существуют средства для описания *сложных* типов данных, основой которых являются описанные в этой главе простые типы данных. Подробному обсуждению сложных типов данных посвящена глава 13.

Итоги

- Программа на ассемблере, отражая особенности архитектуры процессора, состоит из сегментов — блоков памяти, допускающих независимую адресацию.
- Каждый сегмент может состоять из предложений языка ассемблера четырех типов: команд ассемблера, макрокоманд, директив ассемблера и строк комментариев.
- Формальное описание синтаксиса языков программирования, в том числе ассемблера, удобно выполнять с использованием таких метасинтаксических языков, как синтаксические диаграммы и нормальные формы Бэкуса-Наура. Синтаксические диаграммы более наглядны, а расширенные формы Бэкуса-Наура более компактны.
- Предложения ассемблера формируются из *лексем,* представляющих собой синтаксически неразделимые последовательности допустимых символов языка, имеющие смысл для транслятора.
- Ассемблер допускает большое разнообразие типов операндов, которые могут задаваться неявно или содержаться непосредственно в команде, в регистрах и в памяти. В двухоперандной машинной команде возможны следующие сочетания операндов:
  - регистр — регистр;
  - регистр — память;
  - память — регистр;
  - непосредственный операнд — регистр;
  - непосредственный операнд — память.
- Операндами в команде могут быть числа, регистры, ячейки памяти, символьные идентификаторы. При необходимости операнд может быть задан выражением.
- Ассемблер позволяет организовать гибкую *прямую и косвенную* адресацию операндов в памяти.
- Исходный текст программы разбивается на сегменты с помощью директив сегментации, которые делятся на стандартные и упрощенные.
- Упрощенные директивы сегментации позволяют унифицировать интерфейс с языками высокого уровня и облегчают разработку программ, повышая наглядность кода.
- Транслятор TASM поддерживает разнообразные типы данных, которые делятся на простые (базовые) и сложные. Простые типы служат основой для построения сложных типов данных.

- Директивы описания простых типов данных позволяют резервировать и при необходимости инициализировать области памяти заданной длины.
- Доступ к данным в памяти должен производиться с учетом принципа их размещения процессорами IA-32 — «младший байт по младшему адресу».
- Каждой переменной, объявленной с помощью директивы описания данных, ассемблер назначает атрибуты, доступ к которым можно получить с помощью соответствующих операторов ассемблера.

Глава 6
Первая программа

- Жизненный цикл программы на ассемблере
- Пример простой программы
- Разработка программ на ассемблере с использованием пакета TASM
- Разработка программ на ассемблере с использованием пакета MASM
- Трансляция и компоновка программы
- Назначение и структура выходных файлов, формируемых транслятором
- Отладка программ
- Менеджер проекта — утилита MAKE

В этой главе мы, во-первых, познакомимся с процессом разработки программ на ассемблере, во-вторых, научимся использовать специальные программные средства, предназначенные для преобразования исходных текстов на ассемблере в машинные коды для их исполнения компьютером.

Жизненный цикл программы

Процесс разработки программы на ассемблере, включая постановку задачи, получение первых результатов и дальнейшее сопровождение программы, мало чем отличается от традиционного подхода с использованием любого из существующих языков программирования. После адаптации основных положений этого процесса к особенностям ассемблера получится вполне стандартный перечень работ.

1. *Постановка и формулировка* задачи:
   - изучение предметной области и сбор материала в проблемно-ориентированном контексте;
   - определение назначения программы, выработка требований к ней и представление требований, если возможно, в формализованном виде;
   - формулирование требований к представлению исходных данных и выходных результатов;
   - определение структур входных и выходных данных;
   - формирование ограничений и допущений на исходные и выходпые данные.
2. Этап *проектирования:*
   - формирование «ассемблерной» модели задачи;
   - выбор метода реализации задачи;
   - разработка алгоритма реализации задачи;
   - разработка структуры программы в соответствии с выбранной моделью памяти.
3. Этап *кодирования:*
   - уточнение структуры входных и выходных данных и определение ассемблерного формата их представления;
   - программирование задачи;
   - комментирование текста и составление предварительного описания программы.
4. Этап *отладки и тестирования:*
   - составление тестов для проверки работоспособности программы;
   - обнаружение, локализация и устранение в программе ошибок, выявленных в тестах;
   - корректировка кода программы и ее описания.
5. Этап *эксплуатации и сопровождения:*
   - настройка программы на конкретные условия использования;
   - обучение пользователей работе с программой;
   - организация сбора сведений о сбоях в работе программы, ошибках в выходных данных, пожеланиях по улучшению интерфейса и удобства работы с программой;
   - модификация программы с целью устранения выявленных ошибок и, при необходимости, изменения ее функциональных возможностей.

Порядок и объем работ в приведенном перечне укладываются в понятие *жизненного цикла* программы на ассемблере. На практике к порядку применения и иолноте выполнения перечисленных этапов нужно подходить разумно и творчески. Многое определяется особенностями конкретной задачи, ее назначением, объемами кода и обрабатываемых данных, другими характеристиками исходной задачи. Некоторые из этих этапов могут либо выполняться одновременно с другими, либо

вовсе отсутствовать. Главная цель формирования подобного списка работ — в том, чтобы изначально упорядочить процесс создания нового программного продукта с сохранением концептуальной целостности постановки задачи и исключением анархии в процессе разработки.

Пример простой программы

В данном разделе рассмотрим пример простой, но полноценной программы на ассемблере. В главе 2 при обсуждении архитектуры было перечислено большое количество регистров. Как правило, большинство из них задействовано практически в любой программе. Было бы интересно во время работы программы посмотреть (получить) их содержимое. Это нетрудно, если использовать специальную программу — отладчик. Но как сделать это динамически и автономно (без помощи других программ)? Или, к примеру, как решить обратную задачу — ввести некоторое значение с клавиатуры в регистр? Эта и подобная ей задачи являются задачами преобразования данных. Они довольно часто возникают на практике. Причина здесь в том, что компьютер воспринимает данные только тех типов, которые поддерживаются его системой команд. Поэтому на практике часто возникает необходимость преобразования данных из одного представления в другое. В этой главе в качестве примера рассмотрим частный случай решения одной из таких задач — задачи преобразования шестнадцатеричного числа из двух цифр, вводимого с клавиатуры (то есть в символьном виде), в двоичное число. После выполнения этой операции полученное число можно использовать как операнд в двоичных арифметических операциях.

Для начала нужно продумать алгоритм и изучить предметную область. Ввод информации с клавиатуры и вывод ее на экран осуществляются в символьном виде. Кодирование этой информации производится согласно *таблице ASCII.* В таблице ASCII каждый символ кодируется одним байтом. Если работа происходит с числами, то при попытке их обработать сразу возникает проблема: команд для арифметической обработки чисел в символьном виде нет. Что делать? Выход очевиден: нужно преобразовать символьную информацию в формат, поддерживаемый машинными командами. После такого преобразования нужно выполнить необходимые вычисления и преобразовать результат обратно в символьный вид. Затем следует отобразить информацию на мониторе.

Рассматриваемая в этом разделе программа является одним из вариантов решения задачи. В основу ее алгоритма положена особенность, связанная с ACSII-кодами символов соответствующих шестнадцатеричных цифр. Шестнадцатеричные цифры формируются из символов 0, 1, ..., 9, A, B, C, D, E, F, a, b, c, d, e, f, например: 12Af, 34ad. В таблице ASCII можно найти значения ASCII-кодов, соответствующие этим символам. На первый взгляд, непонятна популярность способа представления информации в виде шестнадцатеричных чисел. Из предыдущих глав известно, что аппаратура компьютера построена на логических микросхемах и работает только с двоичной информацией. Если нам требуется проанализировать, например, состояние некоторой области памяти, то разобраться в нагромождении нулей и единиц будет очень непросто. Для примера рассмотрим двоичную последовательность

010100010101011110101101110101010101000101001010.

Это представление не очень наглядно. В главе 2 отмечалось, что оперативная память состоит из ячеек — байтов — по 8 битов. Приведенная выше цепочка битов при разбиении ее на байты будет выглядеть так:

01010001 01010111 10101101 11010101 01010001 01001010.

С наглядностью стало лучше, но если исследуемая область памяти окажется больше, то разобраться будет все равно сложно. Проведем еще одну операцию: каждый байт разобьем на две части по 4 бита — *тетрады*:

0101 0001 0101 0111 1010 1101 1101 0101 0101 0001 0100 1010.

И вот тут проявляется замечательное свойство шестнадцатеричных чисел — каждой тетраде можно поставить в соответствие одну шестнадцатеричную цифру (табл. 6.1).

Таблица 6.1. Кодировка шестнадцатеричных цифр

Шестнадцатеричная цифра	**ASCII-код (двоичноепредставление)**	Двоичная тетрада
0	30h (0011 0000)	0000
1	31h (OOH 0001)	0001
2	32h (0011 0010)	0010
3	33h (0011 0011)	ООП
4	34h (0011 0100)	0100
5	35h (0011 0101)	0101
6	36h (0011 0110)	ОНО
7	37h (OOH 0111)	0111
8	38h (0011 1000)	1000
9	39h (0011 1001)	1001
A, a	41h (0100 0001), 61h (0110 0001)	1010
B, b	42h (0100 0010), 62h (0110 0010)	1011
C, c	43h (0100 0011), 63h (0110 0011)	1100
D, d	44h (0100 0100), 64h (0110 0100)	1101
E, e	45h (0100 0101), 65h (0100 0101)	1110
F, f	46h (0100 0110), 66h (0110 0110)	1111

Если заменить в последней полученной строке тетрады соответствующими шестнадцатеричными цифрами, то получим последовательность 51 57 ad d5 51 8a.

Каждый байт теперь наглядно представлен двумя шестнадцатеричными цифрами, что оказывается очень удобным, в частности, при работе с отладчиком.

Вернемся к нашей задаче. Если посмотреть внимательно на представление шестнадцатеричных цифр и их кодировку в ACSII-коде (табл. 6.1), то можно увидеть полезные закономерности. К примеру, ASCII-код нуля (код символа 0) в шестнадцатеричном виде равен 30h. В двоичном представлении 30h записывается как

0011 0000, а соответствующее двоичное число должно быть двоичным представлением нуля 0000 0000. Отсюда следуют некоторые выводы.

Вывод первый. Для шестнадцатеричных цифр 0...9 ACSII-код отличается от соответствующего двоичного представления на 0011 0000, или 30h. Поэтому для преобразования кода символа шестнадцатеричной цифры в соответствующее двоичное число есть два пути:

- выполнить двоичное вычитание: (ACSII-код)h - 30h;
- обнулить старшую тетраду байта *с* символом шестнадцатеричной цифры в ACSII-коде.

Видно, что с шестнадцатеричными цифрами в диапазоне от 0 до 9 все просто. Что касается шестнадцатеричных цифр a, b, c, d, e, f, то здесь ситуация несколько сложнее. Алгоритм должен распознавать эти символы и производить дополнительные действия при их преобразовании в соответствующее двоичное число. Внимательный взгляд на символы шестнадцатеричных цифр и соответствующие им двоичные представления (см. табл. 6.1) говорит о том, для преобразования уже недостаточно простого вычитания или обнуления старшей тетрады. Анализ таблицы ASCII показывает, что символы прописных букв шестнадцатеричных цифр отличаются от своего двоичного эквивалента на величину 37h. Соответствующие строчные буквы шестнадцатеричных цифр отличаются от своего двоичного эквивалента на 67h.

Отсюда следует *вывод второй.* Алгоритм преобразования должен различать Прописные и строчные буквенные символы шестнадцатеричных цифр и корректировать значение ASCII-кода на величину 37h или 67h.

Обратите внимание на то, что после записи значения шестнадцатеричной цифры следует символ «h». Как упоминалось в главе 4, это сделано для того, чтобы транслятор мог различить в программе одинаковые по форме записи десятичные и шестнадцатеричные числа.

В листинге 6.1 приведен исходный текст программы, которая решает задачу преобразования двузначного шестнадцатеричного числа в символьном виде в двоичное представление.

Листинг 6.1. Пример программы на ассемблере

```
;------Prg_6_1.asm---------------------
;Программа преобразования двузначного шестнадцатеричного числа
;в символьном виде в двоичное представление.
;Вход: исходное шестнадцатеричное число из двух цифр,
;вводится с клавиатуры.
;Выход: результат преобразования помещается в регистр dl.
;--------------------------------------
;--------------------------------------
data segment para public "data"      ;сегмент данных
message db  "Введите две шестнадцатеричные цифры,$"
data ends
stk segment stack
    db 256 dup ("?")                 ;сегмент стека
stk ends
code segment para public "code"      ;начало сегмента кода
main proc                            ;начало процедуры main
    assume cs:code,ds:data,ss:stk
    mov ax,data                     ;адрес сегмента данных в регистр ax
```

продолжение ⇨

Листинг 6.1 *(продолжение)*

```
<19>        mov ds,ax        ;ax в ds
<20>        mov ah,9
<21>        raov dx,offset message
<22>        int 21h
<23>        xor ax,ax        ;очистить регистр ax
<24>        mov ah,1h        ;1h в регистр ah
<25>        int 21h          ;генерация прерывания с номером 21h
<26>        mov dl,al        ;содержимое регистра al в регистр dl
<27>        sub dl,30h       ;вычитание: (dl)=(dl)-30h
<28>        cmp dl,9h        ;сравнить (dl) с 9h
<29>        jle M1           ;перейти на метку M1, если dl<9h или dl=9h
<30>        sub dl,7h        ;вычитание: (dl)=(dl)-7h
<31>    M1:                  ;определение метки M1
<32>        mov cl,4h        ;пересылка 4h в регистр cl
<33>        shl dl,cl        ;сдвиг содержимого dl на 4 разряда влево
<34>        int 21h          ;вызов прерывания с номером 21h
<35>        sub al,30h       ;вычитание: (dl)=(dl)-30h
<36>        cmp al,9h        ;сравнить (al) с 9h
<37>        jle M2           ;перейти на метку M2, если al<9h или al=9h
<38>        sub al,7h        ;вычитание: (al)=(al)-7h
<39>    M2:                  ;определение метки M2
<40>        add dl,al        ;сложение: (dl)=(dl)+(al)
<41>        mov ax,4c00h     ;пересылка 4c00h в регистр ax
<42>        int 21h          ;вызов прерывания с номером 21h
<43>    main endp            ;конец процедуры main
<44>    code ends            ;конец сегмента кода
<45>    end main             ;конец программы с точкой входа main
```

В предыдущих главах говорилось, что процессор аппаратно поддерживает шесть адресно-независимых областей памяти: сегмент кода, сегмент данных, сегмент стека и три дополнительных сегмента данных. Наша программа использует только первые три из них.

Строки 9–11 определяют сегмент данных. В строке 10 описана текстовая строка с сообщением «Введите две шестнадцатеричные цифры».

Строки 12-14 описывают сегмент стека, который является просто областью памяти длиной 256 байт, инициализированной символами «"?"». Отличие сегмента стека от сегментов других типов состоит в использовании и адресации памяти. В отличие от сегмента данных (наличие которого необязательно, если программа не работает с данными), сегмент стека желательно определять всегда.

Строки 15-44 содержат сегмент кода. В этом сегменте в строках 16-43 определена одна процедура main.

Строка 17 содержит директиву ассемблера, которая связывает сегментные регистры с именами сегментов.

Строки 18-19 выполняют инициализацию сегментного регистра DS.

Строки 20-22 выводят на экран сообщение message:

```
Введите две шестнадцатеричные цифры
```

Строка 23 подготавливает регистр AX к работе, обнуляя его. Содержимое AX после этой операции следующее:

ax = 0000 0000 0000 0000

Строки 24-25 обращаются к средствам операционной системы для ввода символа с клавиатуры. Введенный символ операционная система помещает в регистр AL. К примеру, пусть в ответ на сообщение с клавиатуры были введены две шестнадцатеричные цифры:

5C

В результате после отработки команды в строке 25 появится один символ в ASCII-коде — 5, и состояние регистра AX станет таким:

ax = 0000 0001 0011 0101

Строка 26 пересылает содержимое AL в регистр DL. Это делается для того, чтобы освободить AL для ввода второй цифры. Содержимое регистра DX после этой пересылки следующее:

dx = 0000 0000 0011 0101

Строка 27 преобразует символьную цифру 5 в ее двоичный эквивалент путем вычитания 30h, в результате чего в регистре DL будет двоичное значение числа 5:

dx = 0000 0000 0000 0101

В строках 28-29 выясняется, нужно ли корректировать двоичное значение в DL. Если оно лежит в диапазоне 0...9, то в DL находится правильный двоичный эквивалент введенного символа шестнадцатеричной цифры. Если значение в DL больше 9, то введенная цифра является одним из символов A, B, C, D, E, F (строчные буквы для экономии места обрабатывать не будем). В первом случае строка 29 передаст управление на метку M1. При обработке цифры 5 это условие как раз выполняется, поэтому происходит переход на метку M1 (строка 31).

Каждая шестнадцатеричная цифра занимает одну тетраду. У нас две таких цифры, поэтому нужно их разместить так, чтобы старшинство разрядов сохранялось. В строках 32-33 значение в DL сдвигается на 4 разряда влево, освобождая место в младшей тетраде под младшую шестнадцатеричную цифру.

В строке 34 в регистр AL вводится вторая шестнадцатеричная цифра C (ее ASCII-код — 63h):

ax = 0000 0001 0100 0011

В строках 35-37 выясняется, попадает ли двоичный эквивалент второго символа шестнадцатеричной цифры в диапазон 0...9. Наша вторая цифра не попадает в диапазон, поэтому для получения правильного двоичного эквивалента нужно произвести дополнительную корректировку. Это делается в строке 38. Состояние AL после выполнения строки 35 следующее:

ax = 0000 0001 0001 0011

В AL записано число 13h, а нужно, чтобы было 0Ch (помните о правилах записи шестнадцатеричных чисел!). Так как 0Ch не попадает в диапазон 0...9, то происходит переход на строку 38. В результате работы команды вычитания в регистре AL получается правильное значение (al) = 0Ch:

ax = 0000 0001 0000 1100

И наконец, в строке 40 сдвинутое значение в DL складывается с числом в AL:

```
dx  =   0000 0000 0101 0000
    +
ax  =   0000 0001 0000 1100
    =
dx  =   0000 0000 0101 1100
```

Таким образом, в регистре DL получен двоичный эквивалент двух введенных символов, изображающих двузначное шестнадцатеричное число:

(dl) = 05Ch

Строки 41–42 предназначены для завершения программы и возврата управления операционной системе.

Этот пример не случайно рассмотрен столь детально. Он отражает многие специфические особенности программирования на ассемблере. На его основе построен материал следующего раздела, посвященный тому, как из исходного модуля получить правильно функционирующий исполняемый модуль.

Процесс разработки программы

На рис. 6.1 приведена общая схема процесса разработки программы на ассемблере. Название программы соответствует рассмотренному ранее примеру программы (см. листинг 6.1). На схеме выделено четыре этапа этого процесса. На первом этапе, когда вводится код программы, можно использовать любой текстовый редактор. В Windows таким редактором может быть Блокнот (Notepad). При выборе редактора нужно учитывать, что он не должен вставлять «посторонних» символов (специальных символов форматирования). С этой точки зрения Microsoft Word в качестве основного редактора ассемблерных программ не годится. Очень интересный редактор — Asm Editor for Windows (http://www.avtlab.ru). Созданный с помощью текстового редактора файл должен иметь расширение .asm.

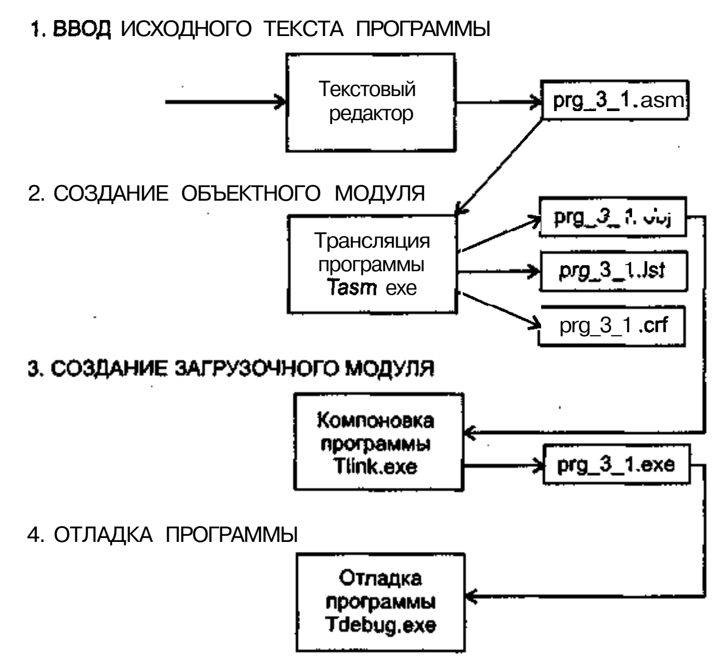


Рис. 6.1. Схема процесса разработки программы на ассемблере

Для выполнения остальных этапов разработки требуются специализированные программные средства из пакета MASM или TASM. В ходе настоящего обсуждения будут описываться средства обоих пакетов, но в основном на примере TASM, поскольку процесс разработки ассемблерных Программ с использованием этого

пакета более нагляден. В принципе, все пакеты ассемблера выполняют практически одну работу, но по-разному, например, маскируют ее с помощью интегрированный среды или объединяют некоторые этапы разработки. Поняв суть преобразований исходной программы, выполняемых пакетом TASM, освоить другие пакеты ассемблера будет на порядок легче и.

Трансляция программы

Следующий шаг на пути создания исполняемого модуля — *трансляция* программы. Для трансляции нужен подготовленный и записанный на диск исходный текст программы (см. листинг 6.1).

На этапе трансляции решается несколько задач:

- перевод команд ассемблера в соответствующие машинные команды;
- построение таблицы символов;
- расширение макросов;
- формирование файла листинга и объектного модуля.

Программа, которая реализует эти задачи, называется *ассемблером.* Итог работы ассемблера — два файла: файл объектного модуля и файл листинга.

Объектный модуль включает в себя представление исходной программы в машинных кодах и некоторую другую информацию, необходимую для отладки и компоновки его с другими модулями. При использовании пакета TASM получение объектного модуля исходного файла производится программой (ассемблером) tasm.exe. Формат командной строки для запуска tasm.exe следующий:

```
TASM [ключи] имя_исходного_файла [,имя_объектного_файла]
[,имя_файла_листинга] [,имя_файла_перекрестных_ссылок]
```

Помнить формат командной строки tasm.exe необязательно. Получить быструю справку о нем на экране можно, если запустить tasm.exe без параметров. Обратите внимание на то, что большинство параметров заключено в квадратные скобки. Это общепринятое соглашение по обозначению необязательных параметров. Из этого следует, что обязательным аргументом командной строки является лишь имя исходного файла. Этот файл должен находиться на диске и обязательно иметь расширение .asm. За именем исходного файла через запятую могут следовать необязательные параметры, обозначающие имена объектного файла, файла листинга и файла перекрестных ссылок. Если не задать их, то соответствующие файлы попросту не будут созданы. Если же их нужно создать, то необходимо учитывать некоторые моменты.

- Если имена объектного файла, файла листинга и файла перекрестных ссылок должны совпадать с именем исходного файла (наиболее типичный случай), то нужно просто поставить запятые вместо имен этих файлов:

  ```
  tasm.exe prg_6_l , , ,
  ```

  В результате будут созданы файлы с одинаковыми именами и разными расширениями, как показано на рис. 6.1 для второго этапа.
- Если имена объектного файла, файла листинга и/или файла перекрестных ссылок не должны совпадать с именем исходного файла, то нужно в командной строке указать имена нужных файлов в соответствующем порядке, к примеру:

```
tasm.exe prg_6_1 , ,prg_list,
```

В результате на диске будут созданы файлы prg_6_1.obj, prg_list.lst, prg_6_1.crf.

- Если требуется выборочное создание файлов, то вместо ненужных файлов необходимо подставить параметр nul. Например:

```
tasm.exe prg_6_1 , ,nul,
```

В результате на диске будут созданы файлы prg_6_1.obj, prg_6_1.crf.

Необязательный аргумент [ключи] позволяет задавать режим работы транслятора TASM. Этих ключей достаточно много, и все они описаны в приложении В (http://www.piter.com/download).

Перед работой с программой tasm.exe желательно провести некоторые подготовительные операции. После установки пакета TASM в каталоге \TASM\BIN, где находится файл tasm.exe, присутствует большое количество файлов. Можно запустить программу tasm.exe прямо отсюда, но тогда созданные ею файлы объектного кода, листинга и перекрестных ссылок тоже окажутся в этом каталоге. Если вы собирается написать всего одну программу, то неудобство не столь заметно, но при работе с несколькими программами очень скоро этот каталог станет похожим на свалку. Чтобы избежать подобной ситуации, рекомендуется выполнить описанную далее процедуру.

1. Создайте в каталоге ..\TASM вложенные каталоги ..\WORK и ..\PROGRAM. Каталог ..\PROGRAM будет использоваться для хранения отлаженных кодов программ и их исполняемых модулей (файлов с расширением .exe). Каталог ..\WORK станет рабочим — в нем будут находиться необходимые для получения исполняемого модуля файлы из пакета транслятора TASM и файл исходного модуля, с которым в данный момент ведется работа. После устранения ошибок в исходном модуле его вместе с исполняемым модулем можно будет переписать в каталог ..\PROGRAM, а из каталога ..\WORK удалить все ненужные файлы, чтобы подготовить его для работы со следующим исходным модулем на ассемблере. Таким образом, в каталоге ..\WORK всегда будет находиться рабочая версия программы, а в каталоге ..\PROGRAM — отлаженная версия.
2. Поместите в каталог ..\WORK файлы tasm.exe, tlink.exe и rtm.exe. Если в дальнейшем в каталоге ..\WORK не окажется каких-то рабочих файлов вашей программы, программы tasm.exe и tlink.exe выдадут соответствующее сообщение.
3. Поместите в каталог ..\WORK файл prg_6_1.asm.

После всех этих действий можно начинать работу. Для этого следует перейти в каталог ..\WORK и инициировать трансляцию программы prg_6_1.asm командной строкой вида

```
tasm.exe /zi prg_6_1 , , .
```

В результате на экране появится последовательность строк. Самая первая из них будет содержать информацию о номере версии пакета TASM, который использовался для трансляции данной программы. В следующей строке будет выведено имя транслируемого файла. Если программа содержит ошибки, то транслятор выдаст на экран строки сообщений, начинающиеся словами «Error» и «Warning». Хотя программа из листинга 6.1 синтаксически правильная, в учебных целях в нее можно внести какую-нибудь бессмыслицу и посмотреть, что получится. Наличие строки со словом «Error» будет говорить о том, что в программе есть недопустимые с точ-

ки зрения синтаксиса комбинации символов. Логика работы программы для транслятора не имеет никакого значения. Можно написать абсолютную чушь, но если она синтаксически правильна, транслятор поспешит обрадовать пользователя, сообщив ему об отсутствии ошибок. Наличие строки «Warning» означает, что конструкция синтаксически правильна, но не соответствует некоторым соглашениям языка, и это может служить источником ошибок в будущем.

Для устранения ошибок нужно определить место их возникновения и проанализировать ситуацию. Местоположение ошибки легко определяется по значению в скобках в сообщении об ошибке. Это значение является номером ошибочной строки. Запомнив его, нужно перейти в файл с исходной программой и по номеру строки найти место ошибки. Этот способ локализации ошибок имеет недостатки. Во-первых, он не нагляден. Во-вторых, не всегда номера строк в сообщении соответствуют действительным номерам ошибочных строк в исходном файле. Такая ситуация будет наблюдаться, например, при использовании макрокоманд. В этом случае транслятор вставляет в файл дополнительные строки в соответствии с описанием применяемой макрокоманды, а в результате происходит сбой в нумерации. По этим соображениям для локализации ошибок лучше использовать информацию из специального создаваемого транслятором *файла листинга*. Этот файл имеет расширение .lst, а его имя определяется в соответствии с рассмотренными выше соглашениями. В листинге 6.2 приведен полный формат файла листинга для программы, содержащей некоторые ошибки. Файл листинга — это всегда очень важный документ, и ему нужно уделить должное внимание.

Листинг 6.2. Пример файла листинга

```
Turbo Assembler  Version 4.102/03/98 21:23:43   Page 1
Prg_6_1.asm
1   ;———Prg_6_1.asm——————————
2   ;Программа преобразования двузначного шестнадцатеричного числа
3   ;в символьном виде в двоичное представление.
4   ;Вход: исходное шестнадцатеричное число из двух цифр,
5   ;вводится с клавиатуры.
6   ;Выход: результат помещается
7   ;в регистр al.
8   ;————————————————————
9   0000data segment para public "data"  ;сегмент данных
10  0000 82 A2 A5 A4 A8 E2 A5+ message db "Введите две шестнадцатеричные
    цифры,$"
11  20 A4 A2 A5 20 E8   A5+
12  E1 E2 AD A0 A4 E6   A0+
13  E2 A5 E0 A8 E7 AD   EB+
14  A5 20 E6 A8 E4 E0   EB+
15  2C 24
16  0025 data ends
17  0000 stk  segment   stack
18  0000 0100*(3F)  db  256 dup ("?")       ;сегмент стека
19  0100 stkends
20  0000 code   segment para public "code"  ;начало сегмента кода
21  0000 main   proc                        ;начало процедуры main
22  assume  cs:code,ds:data,ss:stk
23  0000 B8 0000s  mov ax,data         ;адрес сегмента данных в регистре ax
24  0003 8E D8  mov ds,ax                    ;ax в ds
25  0005 B4 09  mov ah,9
26  0007 BA 0000mov dx,offset messag
**Error** Prg_6_1.asm(21) Undefined symbol: MESSAG
27  000A CD 21  int 21h
```

продолжение ☞

Листинг 6.2 *(продолжение)*

```
28  000C 33 C0   xor ax,ax    ;очистить регистр ax
29  000E B4 01   mov ah,1h    ;1h в регистр ah
30  0010 CD 21   int 21h      ;генерация прерывания с номером 21h
31  0012 8A D0   mov dl,al    ;содержимое регистра al в регистр dl
32  0014 80 EA 30   sub dl,30h   ;вычитание: (dl)=(dl)-30h
33  0017 80 FA 09   cmp dl,9h   ;сравнить (dl) с 9h
34  001A 7E E4   jle MM         ;перейти на метку M1, если dl<9h или dl=9h
**Error** Prg_6_1.asm(29) Undefined symbol: MM
35  001C 80 EA 00   sub dl,777h ;вычитание: (dl)=(dl)-7h
**Error** Prg_6_1.asm(30) Constant too large
36  001FM1:                    ;определение метки M1
37  001F B1 04  mov cl,4h    ;пересылка 4h в регистр cl
38  0021 D2 E2   shl dl,cl    ;сдвиг содержимого dl на 4 разряда влево
39  0023 CD 21  int 21h      ;вызов прерывания с номером 21h
40  0025 2C 30  sub al,30h   ;вычитание: (dl)=(dl)-30h
41  0027 3C 09  cmp al,9h   ;сравнить (al) с 9h 28
42  0029 7E 02  jle M2      ;перейти на метку M2, если al<9h или al=9h
43  002B 2C 07  sub al,7h    ;вычитание: (al)=(al)-7h
44  002DM2:                    ;определение метки M2
45  002D 02 D0  add dl,al    ;сложение: (dl)=(dl)+(al)
46  002F B8 4C00mov ax,4c00h;пересылка 4c00h в регистр ax
47  0032 CD 21  int 21h      ;вызов прерывания с номером 21h
48  0034 main   endp         ;конец процедуры main
49  0034 code   ends         ;конец сегмента кода
50  end main                 ;конец программы с точкой входа main

Turbo Assembler  Version 4.1 02/03/98 21:23:43   Page 2
Symbol Table
Symbol Name TypeValue   Cref(defined at #)
??DATE   Text  "02/03/98"
??FILENAME  Text"Prg_6_1"
??TIME      Text"21:23:43"
??VERSION   Number 040A
@CPU    Text0101H
@CURSEG TextCODE    #9 #17 #20
@FILENAME   TextPRG_6_1
@WORDSIZE   Text2   #9 #17 #20
M1  NearCODE:001F   #36
M2  NearCODE:002D   42 #44
MAIN    NearCODE:0000   #21 50
MESSAGE ByteDATA:0000  #10
Groups & Segments  Bit Size Align  Combine Class   Cref(defined at #)
CODE    16 0034 ParaPublic  CODE#20 22
DATA   16 0025 ParaPublic  DATA#9 22 23
STK 16 0100 ParaStack   17  22
Turbo Assembler  Version 4.1 02/03/98 21:23:43   Page 3
Error Summary
**Error** Prg_6_1.asm(21) Undefined symbol: MESSAG
**Error** Prg_6_1.asm(29) Undefined symbol: MM
**Error** Prg_6_1.asm(30) Constant too large
```

Файл листинга содержит код ассемблера исходной программы, а также расширенную информацию об этом коде. Для каждой команды ассемблера указываются ее машинный (объектный) код и смещение в кодовом сегменте. Кроме того, в конце листинга TASM формирует таблицы с информацией о метках и сегментах, используемых в программе. Если есть ошибки или сомнительные участки кода, то TASM включает в конец листинга сообщения о них. Если сравнить их с сообщениями, выводимыми на экран, то видно, что они совпадают. Кроме того, что очень удобно, эти же сообщения включаются в текст листинга непосредственно после ошибочной строки.

Строки в файле листинга имеют следующий формат:

`глубина_вложенности номер_строки смещение машинный_код исходный_код`

Далее описано каждое из этих полей:

- `глубина_вложенности` — уровень вложенности включаемых файлов или макрокоманд в файле.
- `номер_строки` — номер строки в файле листинга. Номера строк листинга используются для локализации ошибок и формирования *таблицы перекрестных ссылок.*

ПРИМЕЧАНИЕ Как уже упоминалось, номера строк листинга могут не соответствовать номерам строк в исходном файле. В добавление к сказанному ранее нужно отметить, что в ассемблере имеется директива INCLUDE, которая позволяет включить в файл строки другого файла. Нумерация при этом, как и в случае макрокоманд, будет последовательная для строк обоих файлов. Факт вложенности кода одного файла в другой фиксируется увеличением значения поля «глубина_вложенности» на единицу. Это замечание касается и макрокоманд.

- `смещение` — смещение в байтах текущей команды относительно начала сегмента кода. Это смещение называют также *счетчиком адреса.* Величину смещения вычисляет транслятор для адресации в сегменте кода.
- `машинный_код` — машинное представление команды ассемблера, представленной далее в этой строке полем `исходный_код`.
- `исходный_код` — строка кода из исходного файла.

Дальнейшие действия программиста должны зависеть от характера ошибки. По мере накопления опыта ошибки будут происходить чаще всего в результате простых описок. На первых порах особое внимание следует уделять правильности написания синтаксических конструкций, так как ошибки синтаксиса — самые распространенные. Исправив несколько первых ошибок, следует перетранслировать программу и приступить к устранению следующих ошибок. Возможно, что этого делать не придется, так как после исправления одной ошибки могут исчезнуть и последующие (так называемые *наведенные* ошибки).

О нормальном окончании процесса трансляции можно судить по отсутствию строк с сообщениями об ошибках и предупреждениях.

Изучая внимательно файл листинга, следует обратить внимание на то, что не все строки исходной программы имеют соответствующий машинный код (строки 9, 16, 17, 19...22, 48...50). Это обстоятельство обусловлено тем, что исходный файл на ассемблере в общем случае может содержать конструкции следующих типов:

- *команды ассемблера* — конструкции, которым соответствуют машинные команды;
- *директивы ассемблера* — конструкции, которые не генерируют машинных команд, а являются указаниями транслятору на выполнение некоторых действий или служат для задания режима его работы;
- *макрокоманды* — конструкции, которые, будучи представлены одной строкой в исходном файле программы, после обработки транслятором генерируют

в объектном модуле последовательность команд, директив или макрокоманд ассемблера.

Формат файла листинга и его полнота не являются жестко регламентированными. Их можно **изменить,** задавая в исходном файле программы *директивы управления листингом* (приложение Г, http://www.piter.com/download).

Компоновка программы

После устранения ошибок и получения объектного модуля можно приступать к следующему этапу — созданию исполняемого (загрузочного) модуля, или, как еще называют этот процесс, к *компоновке программы.* Главная цель этого этапа — преобразовать код и данные в объектных файлах в их *перемещаемое выполняемое отображение.* Чтобы понять, в чем здесь суть, нужно разобраться, зачем вообще разделяют процесс создания исполняемого модуля на два этапа — трансляцию и компоновку. Это сделано намеренно, чтобы можно было объединять вместе модули, написанные на одном и том же или на разных языках. Формат объектного файла позволяет при определенных условиях объединить несколько отдельно оттранслированных исходных модулей в один модуль. При этом в функции компоновщика входит разрешение внешних ссылок (ссылок на внешние процедуры и переменные) в этих модулях. Результатом работы компоновщика является создание загрузочного файла с расширением .exe. После этого операционная система может загрузить такой файл в память и выполнить его.

Полный формат командной строки для запуска компоновщика довольно сложен (в этой и в большинстве следующих глав мы в основном будем использовать упрощенный формат):

```
TLINK [ключи] список_объектных_файлов [,имя_загрузочного_модуля]
[,имя_файла_карты] [,имя_файла_библиотеки] [,имя_файла_определений]
[,имя_ресурсного_файла]
```

Параметры командной строки для запуска компоновщика перечислены далее.

- ключи — необязательные параметры, управляющие работой компоновщика. Список наиболее часто используемых ключей приведен в приложении В (http://www.piter.com/download). Каждому ключу должен предшествовать символ - (дефис) или / (слеш). При задании имен ключей имеет значение регистр символов.
- **список_объектных_файлов** — обязательный параметр, содержащий список компонуемых файлов с расширением .obj. Файлы должны быть разделены пробелами или знаком + (плюс), например:

  ```
  tlink /v prog + mdf + fdr
  ```

  При необходимости имена файлов снабжают указанием пути к ним.
- **имя_загрузочного_модуля** — необязательный параметр, обозначающий имя формируемого загрузочного модуля. Если оно не указано, то имя загрузочного модуля будет совпадать с первым именем в списке имен объектных файлов.
- **имя_файла_карты** — необязательный параметр, наличие которого обязывает компоновщик создать специальный файл с картой загрузки. В ней перечисляются имена, адреса загрузки и размеры всех сегментов, входящих в программу.
- **имя_файла_библиотеки** — необязательный параметр, который представляет собой путь к файлу библиотеки (.lib). Этот файл создается и обслуживается спе-

циальной утилитой tlib.exe пакета TASM. Утилита позволяет объединить часто используемые подпрограммы в виде объектных модулей в один файл. В дальнейшем можно просто указывать в командной строке tlink.exe имена нужных для компоновки объектных модулей и файл библиотеки, в котором следует искать эти подпрограммы. Если компонуется Windows-приложение, то на месте параметра имя_файла_библиотеки должно указываться имя библиотеки импорта (глава 16).

- имя_файла_определений — необязательный параметр, который представляет собой путь к файлу определений (.def). Этот файл используется при компоновке Windows-приложений (глава 16).
- имя_ресурсного_файла — необязательный параметр, который представляет собой путь к файлу с ресурсами Windows-приложения (.res). Этот файл используется при компоновке Windows-приложений (глава 16).

Рассмотренный нами формат командной строки используется и для 32-разрядного варианта компоновщика tlink32.exe.

Существует возможность задания параметров командной строки компоновщика в текстовом файле. Для этого нужно создать файл с именем tlink.cfg (tlink32.cfg). При вызове компоновщика tlink.exe с параметром tlink.cfg (tlink32.exe tlink32.cfg) ему будет передано содержимое файла tlink.cfg (tlink32.cfg). Например, текст конфигурационного файла tlink32.cfg для создания исполняемого файла Windows-приложения с отладочной информацией должен выглядеть так:

```
/v
/Twe
```

Так же как и в случае команды tasm.exe, совсем не обязательно запоминать подробно синтаксис команды tlink.exe. Для того чтобы получить список ключей программы tlink.exe, достаточно просто запустить ее без параметров.

Для выполнения нашего примера запустим программу tlink.exe командной строкой вида

```
tlink.exe /v prg_6_1.obj
```

В результате вы получите исполняемый модуль с расширением .exe — prg_6_1.exe.

Получив исполняемый модуль, не спешите радоваться. К сожалению, устранение синтаксических ошибок еще не гарантирует, что программа будет хотя бы запускаться, не говоря уже о ее правильной работе. Поэтому обязательным этапом процесса разработки является *отладка.*

Отладка программы

На этапе отладки в соответствие с алгоритмом проверяется правильность функционирования как отдельных фрагментов кода, так и программы в целом. Но даже успешное завершение отладки еще не является гарантией того, что программа будет работать правильно со всеми возможными исходными данными. Поэтому нужно обязательно провести *тестирование* программы, то есть проверить ее работу на «пограничных» и заведомо некорректных исходных данных. Для этого составляются тесты. Вполне возможно, что результаты тестирования не удовлетворят разработчика программы. В этом случае ему придется вносить поправки в код программы, то есть возвращаться к первому шагу процесса разработки (см. рис. 6.1).

Специфика программ на ассемблере состоит в том, что они интенсивно работают с аппаратными ресурсами компьютера. Это обстоятельство заставляет программиста постоянно отслеживать содержимое определенных регистров и областей памяти. Естественно, что человеку трудно следить за этой информацией с большой степенью детализации. Поэтому для локализации логических ошибок в программах используют специальный тип программного обеспечения — программные *отладчики*.

Отладчики бывают двух типов:

* *интегрированные* отладчики, реализованные в виде интегрированной среды, напоминающей среду для языков высокого уровня (Turbo Pascal, Visual C++ и т. д.);
* *автономные* отладчики, представляющие собой отдельные программы.

Ни один из рассматриваемых нами ассемблеров (MASM, TASM) не имеет своей интегрированной среды, поэтому для отладки написанных на языке ассемблера программ используют либо автономные отладчики, либо отладчики некоторой среды программирования (например, Visual C++). С помощью автономного отладчика можно исследовать работу любой программы, для которой создан исполняемый модуль, независимо от того, на каком языке был написан его исходный текст. Для учебных целей ни один из этих подходов не приемлем, так как требует знаний, которыми начинающий программировать на ассемблере, скорее всего, еще не обладает.

ПРИМЕЧАНИЕ Почему в учебнике много программ для MS-DOS? Ведь времена массового использования этой операционной системы давно прошли. На сегодняшний день у этой ОС остался один, но очень важный аспект ее применения — методический. Опыт показывает, что при обучении любому языку программирования, в том числе ассемблеру, на первом месте должен быть сам язык, а не программные средства поддержки процесса программирования на нем. В этом контексте отметим два момента. Во-первых, начинающему изучать ассемблер легче объяснить принципы построения и работы ассемблерных программ в среде реального режима (MS-DOS), чем защищенного (Windows). По мере накопления практического опыта и теоретических знаний с целью их наращивания можно переходить к работе с более сложными приложениями, в том числе и для операционной системы Windows. Во-вторых, для большинства изучающих язык ассемблера его освоение является промежуточным этапом на пути к реализации некоторой большей задачи. Поэтому подавляющее большинство глав данного учебника посвящено рассмотрению различных групп команд ассемблера, для детального изучения которых вполне достаточно среды реального режима (MS-DOS).

Пакеты TASM и MASM имеют достаточно эффективные инструменты разработки программ для среды MS-DOS, работу с которыми вполне по силам освоить даже начинающему программисту. Конечно, сейчас мало кто пишет программы для среды MS-DOS, поэтому работу с 16-разрядными инструментами пакетов TASM и MASM нужно рассматривать как часть методики обучения. Аналогичные

рассуждения относятся и к средствам отладки. В этой книге для программ TASM реального режима будет использоваться 16-разрядный отладчик Turbo Debugger (TD), разработанный фирмой Borland International. Это наиболее удачный отладчик для ассемблерных программ реального режима. Принципиально важно, что работа с TD прививает навыки, которые наверняка окажутся полезными при работе с другими отладчиками, например с отладчиком cv.exe из пакета MASM.

Отладчик TD представляет собой оконную среду отладки программ на уровне исходного ассемблерного текста. Он позволяет решить две главные задачи:

- определить место логической ошибки;
- определить причину логической ошибки.

Перечислим некоторые возможности TD:

* трассировка программы в прямом направлении, то есть последовательное выполнение программы, при котором за один шаг выполняется одна машинная инструкция;
* трассировка программы в обратном направлении, то есть выполнение программы по одной команде за один шаг, но в обратном направлении;
- просмотр и изменение состояния аппаратных ресурсов процессора во время трассировки.

Эти действия позволяют определить место и источник ошибки в программе. Нужно сразу оговориться, что TD не позволяет вносить исправления в исходный текст программы. Однако после определения причины ошибочной ситуации можно, не завершая работу отладчика, внести исправления прямо в машинный код и снова запустить программу. Поскольку после завершения работы отладчика эти изменения не сохраняются, на практике подобное прямое внесение изменений в код не применяют. Изменения вносят в исходный текст программы, заново создают загрузочный модуль, который снова загружают в отладчик.

Правильная организация процесса получения исполняемого модуля, пригодного для отладки на уровне исходного текста, рассмотрена ранее. Далее перечислены ключевые моменты этого процесса.

- В исходной программе обязательно должна быть определена метка для первой команды, с которой начнется выполнение программы. Такая метка может быть собственно меткой или, как видно из листинга 6.1, именем процедуры. Имя этой метки обязательно должно быть указано в конце программы в качестве операнда директивы END:

  ```
  end имя_метки
  ```

  В нашем случае эта метка является именем процедуры MAIN.

- Исходный модуль должен быть оттранслирован с ключом /zi:

  ```
  tasm /zi имя_исходного_модуля , , ,
  ```

  Ключ /zi разрешает транслятору сохранить связь символических имен в программе с их смещениями в сегменте кода, что позволяет отладчику выполнять отладку, используя оригинальные имена.

- Редактирование модуля должно быть осуществлено с ключом /v:

  ```
  tlink /v имя_объектного_модуля
  ```

Ключ /v указывает на необходимость сохранения отладочной информации в исполняемом файле.

Запуск отладчика удобнее производить из командной строки с указанием исполняемого модуля отлаживаемой программы:

```
td имя_исполняемого_модуля
```

Кстати, сам файл отладчика td.exe логично также поместить в наш рабочий каталог ..\WORK. Изначально файлы отладчика находятся в каталоге ..\BIN пакета TASM. Если все же файл td.exe и файл исполняемого модуля при запуске будут находиться в разных каталогах, то в командной строке необходимо указать путь к этому модулю, например:

```
td c:\tasm\work\имя_модуля.exe
```

При правильном выполнении перечисленных действий откроется окно отладчика TD под названием Module с исходным текстом программы prg_6_1.asm. Как он здесь оказался, ведь в командной строке для программы td.exe было указано только имя исполняемого модуля? Это как раз и есть результат действия ключей /zi и /v для tasm и tlink соответственно. Их применение позволило сохранить информацию об использовавшихся в коде на ассемблере символических именах. Для полноты эксперимента можно попытаться получить исполняемый модуль без задания этих ключей и проанализировать результат.

Вернемся к окну Module (рис. 6.2). Внимание следует обратить на так называемый *курсор выполнения* (в виде треугольника). Он указывает на первую команду, подлежащую выполнению. Этой команде предшествует имя метки (в нашем случае роль метки выполняет имя процедуры). Это так называемая *точка входа* в программу. Если внимательно посмотреть на конец исходного текста программы, то видно, что это же имя записано в качестве операнда в заключительной директиве END. Это единственный способ сообщить загрузчику ОС о том, где в исходном тексте программы расположена точка входа в нее. В более сложных программах обычно

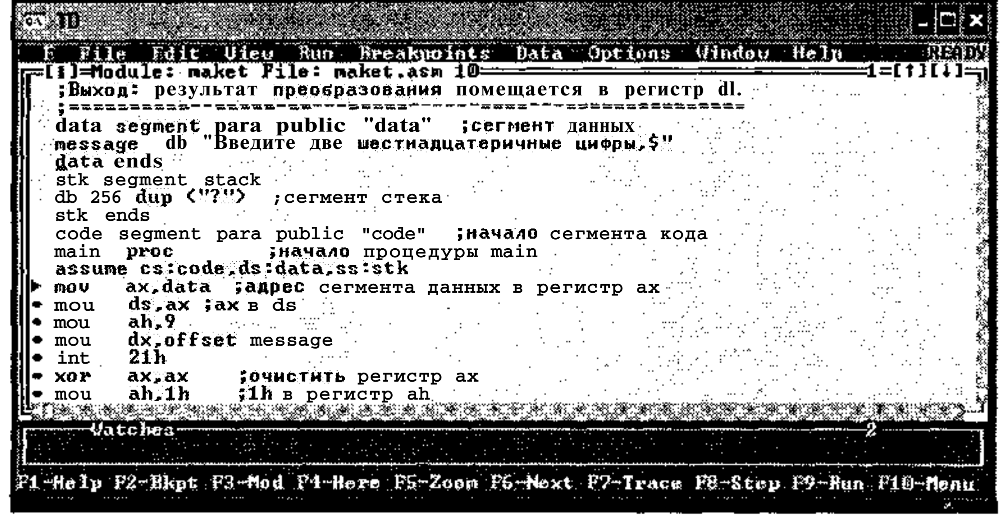


Рис. 6.2. Окно Module отладчика TD

вначале могут идти описания процедур, макрокоманд, и в этом случае без такого явного указания на первую исполняемую команду просто не обойтись.

Основную часть главного окна отладчика обычно занимают одно или несколько дополнительных окон. В каждый момент времени активным может быть только одно из них. Активизация любого окна производится щелчком мышью в любой его видимой точке.

Управление работой отладчика ведется с помощью системы меню. Имеются два типа таких меню:

- *главное меню* — находится в верхней части экрана и доступно постоянно (вызов меню осуществляется нажатием клавиши F10, после чего следует выбрать нужный пункт меню);
- *контекстное меню* — для каждого окна отладчика можно вызвать его собственное меню, которое учитывает особенности этого окна, щелкнув в окне правой кнопкой мыши (либо активизировав окно и нажав клавиши Alt+F10).

Теперь можно проверить правильность функционирования нашей программы.

Специфика программ на ассемблере состоит в том, что делать выводы о правильности их функционирования можно, только отслеживая работу на уровне процессора. При этом нас будет интересовать прежде всего то, как программа использует процессор и изменяет состояние его ресурсов и компьютера в целом.

Запустить программу в отладчике можно в одном из четырех режимов:

- безусловного выполнения;
- выполнения по шагам;
- выполнения до текущего положения курсора;
- выполнения с установкой точек прерывания.

Рассмотрим эти режимы подробнее.

Режим безусловного выполнения программы целесообразно применять, когда требуется посмотреть на общее поведение программы. Для запуска программы в этом режиме необходимо нажать клавишу F9. В точках, где необходимо ввести данные, отладчик, в соответствии с логикой работы применяемого средства ввода, будет осуществлять определенные действия. Аналогичные действия отладчик выполнит при выводе данных. Для просмотра или ввода этой информации можно открыть *окно пользователя* (выбрав в меню команду Window ▸ User screen или нажав клавиши Alt+F5). Если программа работает правильно, то на этом отладку можно и закончить. В случае, если возникают какие-то проблемы или нужно более детально изучить работу программы, применяются три следующих режима отладки.

Режим выполнения программы до текущего положения курсора целесообразно использовать в том случае, если интерес представляет только правильность функционирования некоторого участка программы. Для активизации этого режима необходимо установить курсор на нужную строку программы и нажать клавишу F4. Программа запустится и остановится на отмеченной команде, не выполнив ее. Далее при необходимости вы можете перейти в пошаговый режим.

В *режиме выполнения программы с установкой точек прерывания* программа после запуска будет останавливаться в строго определенных *точках прерывания* (breakpoints). Перед выполнением программы необходимо установить эти точки

в программе, для чего следует перейти к нужной строке и нажать клавишу F2. Выбранные строки подсвечиваются. Установленные ранее точки прерывания можно убрать — для этого нужно повторно перейти к нужной строке и нажать клавишу F2. После установки точек прерывания программа запускается клавишей F9 (см. ранее режим безусловного выполнения). На первой точке прерывания программа остановится. После этого можно посмотреть состояние процессора и памяти, а затем продолжить выполнение программы. Сделать это можно в пошаговом режиме или до следующей точки прерывания.

Режим выполнения программы по шагам применяется для детального изучения ее работы. В этом режиме выполнение программы прерывается на каждой машинной (ассемблерной) команде. При этом становится возможным наблюдение за результатом исполнения команд. Для активизации этого режима нужно нажать клавишу F7 (Run ▸ Trace into) или F8 (Run ▸ Step over). Обе эти клавиши активизируют пошаговый режим; различие их проявляется в том случае, когда в потоке команд встречаются команды перехода в процедуру или на прерывание. При нажатии клавиши F7 отладчик осуществит переход к процедуре или прерыванию и остановится. Если же нажимается клавиша F8, то вызов процедуры или прерывания отрабатывается как одна команда, и управление передается следующей команде программы. Здесь нужно отметить, что кроме окна Module при работе в этом режиме полезно использовать окно CPU, вызвать которое можно через главное меню командой View ▸ CPU.

Окно CPU отражает состояние процессора и состоит из пяти подчиненных окон.

- В окне с исходной программой в дизассемблированном виде представлена та же самая программа, что и в окне Module, но уже в машинных кодах. Пошаговую отладку можно производить прямо в этом окне; строка с текущей командой подсвечивается.
- В окне регистров процессора (Registers) отражается текущее содержимое регистров (по умолчанию — только регистров процессора i8086). Чтобы увидеть регистры i486 или Pentium, нужно задать режим их отображения. Для этого щелкните правой кнопкой мыши в области окна регистров и выберите в контекстном меню команду Registers 32-bit — Yes.
- В окне флагов (Flags) отражается текущее состояние флагов процессора в соответствии с их мнемоническими названиями.
- В окне стека (Stack) отражается содержимое памяти, выделенной для стека. Адрес области стека определяется содержимым регистров SS и SP.
- Окно дампа оперативной памяти (Dump) отражает содержимое области памяти по адресу, который формируется из компонентов, указанных в левой части окна. В окне можно увидеть содержимое произвольной области памяти. Для этого нужно в контекстном меню выбрать нужную команду.

Заметим, что окно CPU, по сути, отражает видимую часть программной модели процессора. Некоторые из подчиненных окон окна CPU можно вывести на экран отдельно. Хотя удобнее работать с исходным текстом в окне Module, чем с его дизассемблированным вариантом в окне CPU, часто есть необходимость отслеживать состояние процессора с помощью подчиненных окон окна CPU. Совместить воз-

можности окон Module и CPU можно, выбрав в меню View имена нужных подчиненныхоконокнаCPU.

Прервать выполнение программы в любом из режимов можно, нажав клавиши Ctrl+F2.

Особенности разработки программ в MASM

Для успешной работы с ассемблером MASM корпорации Microsoft в современных операционных средах (Windows NT\2000\XP) необходимо иметь версию 6.13 этого пакета или выше. В него входят следующие основные программы:

- masm.exe — ассемблер;
- ml.exe — ассемблер и компоновщик (Masm and Link);
- link.exe — компоновщик;
- cv.exe — отладчик (CodeView);
- lib.exe, implib.exe, nmake.exe, cref.exe, h2inc.exe, exehdr.exe, cvpack.exe, helpmake.exe, rm.exe, undel.exe, exp.exe — вспомогательные утилиты.

В пакете MASM изначально предпринимались попытки совместить удобство средств программирования, свойственных языкам высокого уровня, с традиционными средствами машинно-ориентированных языков. Например, буква «M» в названии пакета означает слово «macro», то есть возможность создания *макроопределений* (или *макросов),* представляющих собой именованные группы команд. Их можно вставлять в программу в любом месте, указав только имя группы. Когда-то такая возможность действительно была отличительным свойством этого пакета, но сейчас этим никого не удивишь — макросредства есть во всех ассемблерах для платформы Intel (например, TASM), но, тем не менее, название осталось.

С помощью пакета MASM разработка программ выполняется традиционным для ассемблерного программирования способом — запуском отдельных программ трансляции, компоновки и отладки. Для этого используются программы masm.exe, ml.exe, link.exe и cv.exe. Во избежание путаницы необходимо отметить, что трансляция исходного файла может производиться двумя программами: masm.exe и ml.exe. В чем разница? До выхода MASM версии 5.1 включительно программа masm.exe была самостоятельным транслятором ассемблера. Начиная с MASM версии 6.0 трансляция ассемблерного файла обычно выполняется программой ml.exe, которая кроме трансляции файла вызывает компоновщик link.exe. Это изменение сделано с целью унификации вызовов компиляторов командной строки для всех поддерживаемых этой фирмой языков программирования. К примеру, компиляцию программы на языке C из командной строки выполняет программа cl.exe. Ее задачи — компиляция исходной программы на языках C/C++ и, при отсутствии синтаксических ошибок, компоновка и формирование исполняемого модуля. Аналогичные задачи решает и программа ml.exe. Хотя в пакете MASM 6.13 допустимо использование программы masm.exe, нужно иметь в виду, что запуск link.exe в этом случае также должен выполняться отдельно. Наличие программы masm.exe в пакете MASM 6.13 можно объяснить соображениями совместимости с предыдущими версиями пакета, поэтому особого смысла в ее использовании нет.

Далее приведены форматы командных строк для запуска программ ml.exe, masm.exe и link.exe, а описание их ключей можно найти в приложении В (http://www.piter.com/download).

Командная строка ml.exe имеет вид

```
ml [ключи] исх_файл_1 [[ключи] исх_файл_2] ... [/link ключи_link]
```

Ключи командной строки для ml.exe чувствительны к регистру.

Командная строка masm.exe имеет вид

```
masm [ключи] исх_файл [, [объектный_файл] [, [файл_листинга]
[, [файл_перекрестных_ссылок]]]]
```

Компоновщик компонует (объединяет) объектные файлы и библиотеки в исполняемый файл или динамически компонуемую библиотеку (DLL). Командная строка link.exe имеет вид

```
link [ключи] объект_файлы [,[исполн_файл][,[файл_карты][,[файлы_библиотек]
[,[def_файл]]]]][;]
```

Так же как и в случае пакета TASM, для эффективной работы с MASM нужно провести некоторую дополнительную работу. Исполняемые файлы пакета MASM 6.13 находятся в двух каталогах: ..\BIN и ..\BINR. Для удобства работы их лучше объединить в одном каталоге, к примеру, в каталоге ..\WORK. После этого остается поместить туда исходный файл. Пример командной строки для получения пригодного к отладке исполняемого модуля может быть следующим:

```
ML.EXE /Zi /Fl Prg_6_1.asm
```

Если нет синтаксических ошибок, то можно запускать отладчик:

```
CV.EXE PRG_6_1.EXE
```

Особенности разработки программ в Microsoft Visual Studio

С программами на ассемблере можно работать в среде Visual Studio. Доступны два варианта работы:

- создание и использование команд меню Tools;
- создание полноценного проекта и настройка его параметров.

Эти варианты не равнозначны. Первый вариант означает, по сути, скрытый вызов программ ml.exe и cv.exe. Он хорошо подходит для разработки 16-разрядных программ и является для них единственно возможным. Второй вариант опирается на все возможности интегрированной среды, но подходит только для разработки 32-разрядных программ (Windows-приложений). Мы ограничимся первым вариантом, поскольку второй требует изучения интегрированной среды Microsoft Visual Studio, что не относится к теме данной книги.

Microsoft Visual Studio является удобной средой для создания и редактирования исходных текстов (и только). Что касается самого процесса получения исполняемого модуля, то все действия, которые раньше выполнялись с помощью командной строки, здесь выполняются выбором соответствующей команды в меню Tools. Таким образом, задача сводится к созданию в меню Tools дополнительных команд, предназначенных для обращения к инструментальным средствам пакета ассемблера.

Для того чтобы обсуждение было однозначным, необходимо условиться о том, что все нужные для работы файлы будут расположены, к примеру, в каталоге g:\MASM613\work. В этот рабочий каталог необходимо поместить содержимое каталогов g:\MASM613\BINR и g:\MASM613\BINR, а также файлы программы, с которой в данное время идет работа. Для примера будем работать с программой из листинга 6.1.

После открытия среды Visual C++ 6.0 выполняется следующая последовательность шагов.

1. Выберите команду Tools ▸ Customize и в появившемся диалоговом окне перейдите на вкладку Tools.

   Вкладка Tools позволяет добавлять пункты в меню Tools главного окна. Содержимое данной вкладки состоит из списка Menu contents, ряда текстовых полей и переключателей. Каждый пункт списка Menu contents связан со своими значениями остальных элементов вкладки Tools. Логически эти элементы представляют собой набор параметров для вызова некоторой программы. Чтобы убедиться в этом, достаточно пройтись по нескольким элементам списка. Последний элемент списка обозначен пустым прямоугольником. С его помощью производится добавление новых подпунктов в выпадающее меню Tools главного меню.

2. Чтобы создать команду для вызова программы ml.exe, введите в пустой пункт списка Menu contents название новой команды, например Build ASM Program, и нажмите клавишу Enter.

3. Настроим теперь параметры вызова программы ml.exe. Для этого в поле Command введите g:\MASM613\Work\ml.exe, в поле Arguments — /Zi $(FileName).asm /link /co, в поле Initial directory — $(FileDir) и установите переключатель Use output window.

4. Таким же образом можно добавить в меню Tools команду Debug ASM Program. Для этого в поле Command введите g:\MASM613\Work\cv.exe, в поле Arguments — $(FileName), в поле Initial directory — $(FileDir) и установите переключатель Close window on exiting.

5. Для запуска полученного исполняемого модуля добавим еще одну команду Execute ASM Program. Для этого в поле Command введите C:\WINDOWS\System32\cmd.exe, в полеArguments — /C $(FileName), в поле Initial directory — $(FileDir).

После выполнения этих шагов можно закрыть окно Customize и проверить работоспособность созданных команд меню Tools главного окна. Перед испытанием в среду Visual Studio необходимо загрузить файл с текстом программы prg_6_1.asm (см. листинг 6.1). После открытия файла выберите команду Tools ▸ Build ASM Program. При наличии синтаксических ошибок в окне Output появятся соответствующие сообщения. Для того чтобы перейти к строке исходного текста, вызвавшей недовольство компилятора, достаточно двойного щелчка мышью на строке с текстом ошибки в окне Output. В результате этого курсор в окне с исходным текстом будет установлен на строку с ошибкой.

Если программа ml.exe отработала без ошибок, то полученный с ее помощью исполняемый модуль можно либо сразу запустить, либо предварительно отладить. Запуск производится командой Tools ▸ Execute ASM Program. Для отладки выбирается команда Tools > Debug ASM Program, в результате чего вызывается отладчик cv.exe.

Последний компилятор Visual C++ 7.0 из состава Microsoft Visual Studio Net также имеет настраиваемое меню Tools, поэтому при желании его среду также можно настроить для выполнения описанных ранее задач.

Выбор пакета ассемблера

Какой из пакетов ассемблера выбрать для практической работы? Если этот выбор делать с точки зрения долгосрочной перспективы, то, безусловно, MASM. Причи-

на для этого — продолжающаяся поддержка и развитие этого пакета фирмой Microsoft. Для учебных целей более привлекателен TASM. В данном учебнике выбран промежуточный вариант — транслятор TASM в режиме MASM. Этот режим поддерживает почти все возможности транслятора MASM и создан специально для разработки переносимых (на уровне исходных текстов) программ. Такой подход имеет преимущества в том, что читатель, во-первых, получает навыки использования двух пакетов и, во-вторых, по мере познания особенностей пакетов становится свободным в своем выборе.

Процесс разработки программ с помощью пакетов TASM и MASM предполагает, что пользователь интенсивно работает с командной строкой. При этом он должен помнить не только последовательность запуска различных программ, формирующих исполняемый модуль, но и задаваемые при этом параметры программ. Если проект состоит из большого количества файлов, необходимость помнить всю эту информацию довольно утомительна. Для решения этой проблемы можно посоветовать использовать специальную программу — менеджер проекта. В пакете TASM такой программой является утилита make.exe, а в пакете MASM — утилита nmake.exe. Эти утилиты работают со специально оформленными файлами, называемыми файлами описания, или make-файлами. В make-файлах задаются отношения между файлами проекта и действия над этими файлами, которые выполняются в зависимости от возникновения тех или иных условий. Главный принцип, положенный в основу работы утилиты make.exe, заключается в анализе времени изменения или создания файлов. Подробное описание утилиты можно найти среди файлов к книге[1].

Итоги

- Структура программы на ассемблере отражает особенности архитектуры процессора. Для процессоров Intel типичная программа состоит из трех сегментов: кода, стека и данных. Но это не обязательное условие; например, если программа не использует стек и для ее работы не требуется определения данных, то она может состоять всего лишь из одного сегмента кода.
- Программа на ассемблере работает на уровне аппаратных средств, входящих в программную модель процессора, которая описана в главе 2.
- При разработке алгоритма работы программы и его реализации на ассемблере программист сам должен беспокоиться о размещении данных в памяти, об эффективном использовании ограниченного количества регистров, об организации связи с операционной системой и другими программами.
- Специфика разработки программы на ассемблере состоит в том, что программист должен уделять внимание не только и не столько особенностям моделирования предметной области, сколько тому, как при этом наиболее эффективно и корректно использовать ресурсы процессора.

[1] Все прилагаемые к книге файлы можно найти по адресу http://www.piter.com/download. — *Примеч. ред.*

- В результате работы транслятора создаются файл объектного модуля и файл листинга программы, содержащий разнообразную информацию о программе: объектный код, сообщения о синтаксических ошибках, таблицу символов и т. д. Имея небольшой опыт, из файла листинга можно извлечь массу полезной информации.
- После получения корректного объектного модуля программу необходимо скомпоновать. Для этого применяется утилита-компоновщик, одним из основных назначений которой является разрешение внешних ссылок. Если целевая программа состоит из нескольких отдельно оттранслированных модулей и в них есть взаимные ссылки на переменные или модули, то компоновщик разрешает их, формируя тем самым правильные перемещаемые адреса.
- Результатом работы компоновщика является исполняемый (загрузочный) модуль, имеющий расширение .exe. Его уже можно запускать в надежде, что он правильно выполнит задуманные программистом действия. Но чаще всего при первых запусках программы этого не случается, что говорит о наличии в программе логических ошибок, поиск которых без специальных средств может быть долгим. Для поиска и устранения логических ошибок предназначен специальный вид программного обеспечения — отладчики. С их помощью в большинстве случаев довольно быстро удается снять большинство программных проблем подобного рода.

Глава 7
Команды обмена данными

- **Линейные алгоритмы**
- **Команды пересылки данных**
- **Ввод из порта и вывод в порт**
- **Команды работы с адресами памяти**
- **Команды работы со стеком**

Наверняка вы уже знакомы с понятием алгоритма, представляющего собой формализованное описание логики работы программы. Способы такой формализации весьма разнятся: от текстового описания последовательности действий до алгоритма развитых case-систем. Последовательность действий, описываемых алгоритмом, может быть:

- *линейной* — все действия выполняются поочередно, друг за другом;
- *нелинейной* — в алгоритме есть точки ветвления, в которых должно приниматься решение о месте, с которого программа продолжит свое выполнение, причем решение может носить условный или безусловный характер.

Линейные участки алгоритма обычно содержат команды манипуляции данными, вычисления значений выражений, преобразования данных. В точках ветвления размещают команды сравнения, различных видов перехода, вызова подпрограмм и некоторые другие.

Еще раз обратимся к функциональной классификации целочисленных машинных команд процессора (см. рис. 3.3). Из всей совокупности этих команд на линейных участках работают следующие группы:

- команды пересылки данных;
- арифметические команды;

- логические команды;
- команды управления состоянием процессора.

В этой главе мы рассмотрим только группу команд пересылки данных. Эти команды осуществляют пересылку данных из одного места в другое, запись и чтение информации из портов ввода-вывода, преобразование информации, манипуляции с адресами и указателями, обращение к стеку. Для некоторых из этих команд операция пересылки является только частью алгоритма. Другая его часть выполняет некоторые дополнительные операции над пересылаемой информацией. Поэтому для удобства практического применения и отражения их специфики данные команды будут рассмотрены в соответствии с их функциональным назначением, согласно которому они делятся на команды:

- собственно пересылки данных;
- ввода из порта и вывода в порт;
- работы с адресами и указателями;
- преобразования данных;
- работы со стеком.

Пересылка данных

К группе команд пересылки данных относятся следующие команды:

```
mov <операнд назначения>,<операнд-источник>
xchg <операнд1>,<операнд2>
```

MOV — это основная команда пересылки данных. Она реализует самые разнообразные варианты пересылки. Отметим особенности применения этой команды.

- Командой MOV нельзя осуществить пересылку из одной области памяти в другую. Если такая необходимость возникает, то нужно использовать в качестве промежуточного буфера любой доступный в данный момент регистр общего назначения. К примеру, рассмотрим фрагмент программы для пересылки байта из ячейки fls в ячейку fld:

```
masm
model   small
.data
fls db  5
fld db  1
.code
start:
...
    mov al,fls
    mov fld,al
...
end start
```

- Нельзя- загрузить в сегментный регистр значение непосредственно из памяти. Для такой загрузки требуется промежуточный объект. Это может быть регистр общего назначения или стек. Если вы посмотрите на листинг 5.1, то увидите в начале сегмента кода две команды MOV, выполняющие настройку сегментного регистра DS. При этом из-за невозможности напрямую загрузить в сегментный

регистр значение адреса сегмента, содержащееся в предопределенной переменной @data, приходится использовать регистр общего назначения АХ.

- Нельзя переслать содержимое одного сегментного регистра в другой сегментный регистр. Это объясняется тем, что в системе команд нет соответствующего кода операции. Но необходимость в таком действии часто возникает. Выполнить такую пересылку можно, используя в качестве промежуточных все те же регистры общего назначения. Вот пример инициализации регистра ES значением из регистра DS:

```
mov ax,ds
mov es,ax
```

Но есть и другой, более «красивый» способ выполнения данной операции — использование стека и команд PUSH и POP:

```
push    ds  ;поместить значение регистра ds в стек
pop es  ;записать в es число из стека
```

- Нельзя использовать сегментный регистр CS в качестве операнда назначения. Причина здесь простая. Дело в том, что в архитектуре процессора IA-32 пара CS:IP содержит адрес команды, которая должна выполняться следующей. Изменение командой MOV содержимого регистра CS фактически означало бы операцию перехода, а не пересылки, что недопустимо.

В связи с командой MOV отметим один тонкий момент. Пусть в регистре ВХ содержится адрес некоторого поля (то есть мы используем косвенную базовую адресацию). Его содержимое нужно переслать в регистр АХ. Очевидно, что нужно применить команду MOV в форме

```
mov ax,[bx]
```

Транслятор задает себе вопрос: что адресует регистр ВХ в памяти — слово или байт? Обычно в этом случае он принимает решение сам, по размеру большего операнда, но может и выдать предупреждающее сообщение о возможном несовпадении типов операндов.

Или другой случай — возможно, более показательный. Рассмотрим команды *инкремента* INC (увеличения операнда на 1) и *декремента* DEC (уменьшения операнда на 1):

```
inc [bx]
...
dec [bx]
```

Что адресуется регистром ВХ в памяти — байт, слово или двойное слово?

Допустим также, что требуется инициализировать поле, адресуемое регистром ВХ, значением 0. Очевидно, что одно из решений — применение команды MOV:

```
mov [bx],0
```

И опять у транслятора вопрос: какую машинную команду ему конструировать? Для инициализации байта? Для инициализации слова? Для инициализации двойного слова?

Во всех этих случаях необходимо уточнять тип используемых операндов. Для этого существует специальный оператор ассемблера PTR (см. приложение). Правильно записать приведенные ранее команды можно следующим образом:

```
mov ax,word ptr[bx]  ;если [bx] адресует слово в памяти
inc byte ptr[bx]     ;если [bx] адресует байт в памяти
```

```
dec dword ptr[bx]    ;если [bx] адресует двойное слово в памяти
mov word ptr[bx],0   ;если [bx] адресует слово в памяти
```

Можно рекомендовать использовать оператор PTR во всех сомнительных относительно согласования размеров операндов случаях. Также этот оператор нужно применять, когда требуется принудительно поменять размерность операндов. К примеру, требуется переслать значение 0ffh во второй байт поля flp:

```
masm
model   small
.data
...
flp dw  0
...
.code
start:
...
    mov byte ptr (flp+1),0ffh
...
end start
```

Несмотря на то что поле flp имеет тип WORD, мы сообщаем ассемблеру, что поле нужно трактовать как байтовое, и заставляем вычислить значение эффективного адреса второго операнда как смещение flp плюс единица. Тем самым мы получаем доступ ко второму байту поля flp.

Для двунаправленной пересылки данных применяют команду XCHG. Для этой операции можно, конечно, применить последовательность из нескольких команд MOV, но из-за того что операция обмена используется довольно часто, разработчики системы команд процессора посчитали нужным ввести отдельную команду обмена — XCHG. Естественно, что операнды должны иметь один тип. Не допускается (как и для всех команд ассемблера) напрямую обменивать между собой содержимое двух ячеек памяти. К примеру,

```
xchg    ax,bx   ;обменять содержимое регистров ax и bx
;обменять содержимое регистра ax и слова в памяти по адресу в [si]:
xchg    ax,word ptr [si]
```

Ввод из порта и вывод в порт

В главе 5 при обсуждении вопроса о том, где могут находиться операнды машинной команды, мы упоминали *порт ввода-вывода*. Физически порт ввода-вывода представляет регистр разрядностью 8, 16 или 32 бита. Доступ к устройствам ввода-вывода, системным устройствам компьютера осуществляется посредством этих регистров, причем каждый из этих регистров должен иметь возможность уникальной идентификации. С этой целью архитектурно процессор поддерживает так называемое *адресное пространство ввода-вывода*. Адресное пространство ввода-вывода физически независимо от пространства оперативной памяти и имеет ограниченный объем, составляющий 2^{16}, или 65 536, адресов ввода-вывода.

Таким образом, порт ввода-вывода можно определить как 8-, 16- или 32-разрядный аппаратный регистр, имеющий определенный адрес в адресном пространстве ввода-вывода. Вся работа системы с устройствами на самом низком уровне выполняется с использованием портов ввода-вывода. Посмотрите на рис. 7.1. На нем показана сильно упрощенная концептуальная схема управления оборудованием компьютера.

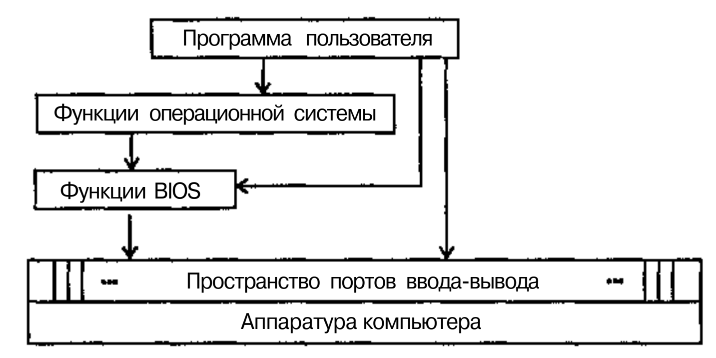


Рис. 7.1. Концептуальная схема управления оборудованием компьютера

Как видно из рисунка, самым нижним уровнем является уровень BIOS, на котором работа с оборудованием ведется напрямую через порты. Тем самым реализуется концепция независимости от оборудования. При замене оборудования потребуется лишь подправить соответствующие функции BIOS, переориентировав их на новые адреса и логику работы портов.

Принципиально управлять устройствами напрямую через порты несложно. Сведения о номерах портов, их разрядности, формате управляющей информации приводятся в техническом описании устройства. Необходимо знать лишь конечную цель своих действий, алгоритм, в соответствии с которым работает конкретное устройство, и порядок программирования его портов. То есть, фактически, нужно знать, что и в какой последовательности нужно послать в порт (при записи в него) или считать из него (при чтении) и как следует трактовать эту информацию. Для этого достаточно всего двух команд, присутствующих в системе команд процессора:

- in **<аккумулятор>,<номер_порта>** — ввод в аккумулятор из порта с номером <номер_порта>;
- out **<номер_порта>,<аккумулятор>** — вывод содержимого аккумулятора в порт с номером **<номер_порта>**.

Возможные значения операндов этих команд приведены в приложении. Необходимо отметить, что использовать эти команды вы сможете без проблем только в программе, предназначенной для MS-DOS. При попытке их запуска в программе для Windows вы получите ошибку. Это не означает невозможности запуска исполняемого модуля описанной далее программы в сеансе Windows. Более того, Windows поддержит реализацию полного цикла разработки данной программы, но сделано это будет в специальном режиме работы — режиме виртуального процессора x86.

В качестве примера рассмотрим, как на уровне аппаратуры заставить компьютер издавать звуки через свой внутренний динамик. На большинстве компьютеров читателей это будет некоторый треск. Изменяя различные параметры программы, в идеале, вы можете получить звук, напоминающий сирену.

Вначале мы перечислим, какие аппаратные ресурсы будут задействованы и как ими надо управлять.

В большинстве компьютеров есть внутренний динамик. Раньше он использовался для того, чтобы издавать звуки при работе самых различный приложений, вплоть до игровых. Сейчас у него осталась единственная важная функция — воспроизведение звуков, которые генерирует BIOS на этапе тестирования и начальной загрузки.

ПРИМЕЧАНИЕ Несмотря на то что прямой доступ к портам ввода-вывода доступен только из среды MS-DOS, сведения о номерах портов и особенностях работы с ними полезны и при программировании для Windows. Если системы Windows 95/98 практически не закрывают доступ к портам, то в Windows NT/2000/XP любая попытка обращения к ним приведет к возникновению ошибки. Причина в том, что порты являются критически важным ресурсом, и механизмы защиты Windows NT/2000/XP не могут допустить их монополизацию каким-либо приложением. Операционная система Windows NT/2000/XP предоставляет программисту функции API для работы с устройствами, посредством которых в конечном итоге и осуществляется доступ к портам посредством команд IN и OUT. Попытка использовать эти команды в программе пользователя в среде Windows NT/2000/XP приведет к возникновению исключения по недопустимому коду операции. Важно понимать, что на низком уровне управление аппаратурой компьютера ведется с использованием тех же портов, что и при работе в MS-DOS, то есть физика остается, меняется логика управления аппаратурой. Если программе удастся получить уровень привилегий ядра (такой уровень имеют драйверы устройств), то в этом случае она может беспрепятственно использовать команды IN и OUT и работать с устройством так же, как в среде MS-DOS. Существуют специальные программы, которые позволяют получить доступ к портам ввода/вывода из программы пользователя, исключая необходимость написания драйвера. Среди файлов, прилагаемых к книге, содержатся две наиболее известные из таких программ — UserPort и PortTalk[1].

Как это ни удивительно, но специальной схемы генерации звука для внутреннего динамика нет. Сигнал для управления динамиком формируется в результате совместной работы следующих микросхем:

- *программируемого периферийного интерфейса* (ППИ) i8255;
- таймера i8253.

Общая схема формирования такого сигнала показана на рис. 7.2.

Обсудим представленную схему. Основная работа по генерации звука производится микросхемой таймера. Микросхема таймера (далее просто таймер) имеет три канала с совершенно одинаковыми внутренней структурой и принципом работы. На каналы таймера подаются импульсы от микросхемы системных часов, которые, по сути, представляют собой генератор импульсов, работающий с частотой 1,19 МГц. Каждый канал имеет два входа и один выход. Выходы канала замкнуты на вполне определенные устройства компьютера. Так, канал 0 замкнут на

[1] Все прилагаемые к книге файлы можно найти по адресу http://www.piter.com/download. — *Примеч. ред.*

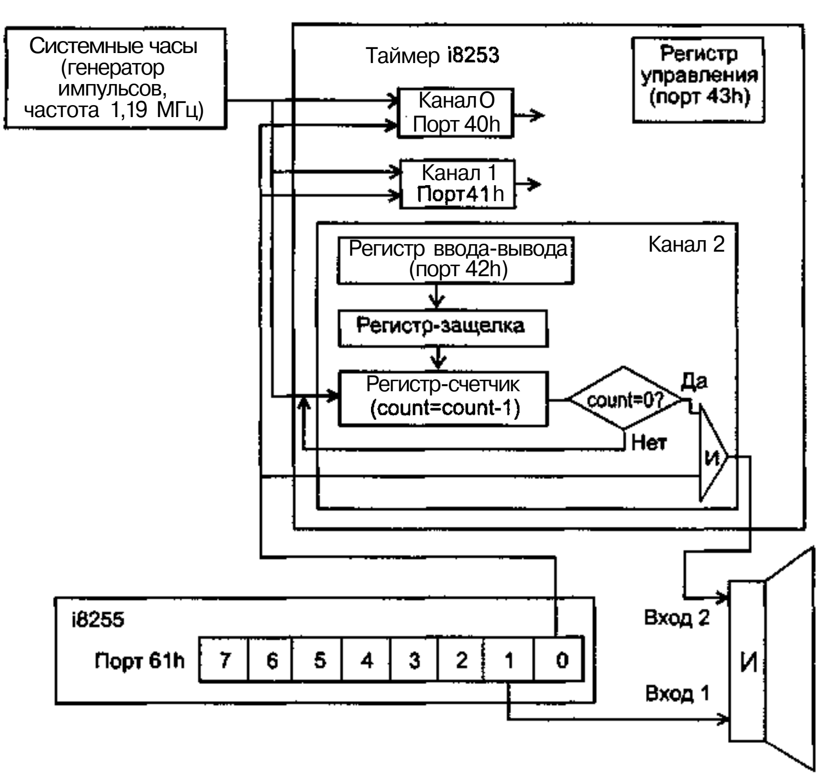


Рис. 7.2. Схема формирования звука для встроенного динамика

контроллер прерываний, являясь источником аппаратного прерывания от таймера, которое возникает 18,2 раза в секунду. Канал 1 связан с микросхемой прямого доступа к памяти (DMA). И наконец, канал 2 выходит на динамик компьютера. Как мы отметили, каналы таймера имеют одинаковую структуру, основу которой составляют три регистра: *регистр ввода-вывода* разрядностью 8 битов, *регистр-фиксатор* (latch register) и *регистр-счетчик* (counter register), оба по 16 битов. Все регистры связаны между собой следующим образом. В регистр ввода-вывода извне помещается некоторое значение. Источником этого значения может быть либо системное программное обеспечение, либо программа пользователя. Каждый регистр ввода-вывода имеет адрес в адресном пространстве ввода-вывода (номер порта ввода-вывода). Регистр ввода-вывода канала 2 имеет номер порта ввода-вывода 42h. Помещаемые в него значения немедленно попадают в регистр-фиксатор, где значение сохраняется до тех пор, пока в регистр ввода-вывода не будет записано новое значение. Но как согласуются эти регистры по их разрядности, ведь один из них 8-, а другой 16-разрядный? Для этого предназначен *регистр управления* (ему соответствует порт 43h), который является частью механизма управления всей микросхемой таймера. Он содержит *слово состояния,* с помощью которого производятся выбор канала, задание режима работы канала и типа операции передачи значения в канал.

Далее описана структура слова состояния:

- Бит 0 определяет тип константы пересчета: 0 — константа задана двоичным числом, 1 — константа задана двоично-десятичным (BCD) числом. Константа пересчета — значение, загружаемое извне в регистр-фиксатор; в нашем случае загружаться будет двоичное число, поэтому значение этого поля будет равно 0.
- Биты 1-3 определяют режим работы микросхемы таймера. Всего можно определить шесть режимов, но обычно используется третий, поэтому для нашего случая значение поля — 011.
- Биты 4-5 определяют тип операции: 00 — передать значение счетчика в регистр-фиксатор (то есть возможны не только операция записи значения в канал, но и извлечение значения регистра-счетчика из него), 10 — записать в регистр-фиксатор только старший байт, 01 — записать в регистр-фиксатор только младший байт, 11 — записать в регистр-фиксатор сначала старший байт, затем младший. В нашем случае значение поля будет 11. Поэтому формирование 16-разрядного регистра-фиксатора через 8-разрядный регистр ввода-вывода производится следующим образом: запись производится в два приема, первый байт из регистра ввода-вывода записывается на место старшего байта регистра-фиксатора, второй байт — на место младшего байта. Нетрудно догадаться, что в регистр ввода-вывода эти байты помещаются командами IN и OUT.
- Биты 6-7 определяют номер программируемого канала. В нашем случае они равны 10.

Для формирования любого звука необходимо задать его длительность и высоту. После того как значение из регистра ввода-вывода попало в регистр-фиксатор, оно моментально записывается в регистр-счетчик. Сразу же после этого значение регистра-счетчика начинает уменьшаться на единицу с приходом каждого импульса от системных часов. На выходе любого из трех каналов таймера стоит схема логического умножения. Эта схема имеет два входа и один выход. Значение регистра-счетчика участвует в формировании сигнала на одном из входов схемы логического умножения И. Сигнал на втором входе этой схемы зависит от состояния бита 0 регистра микросхемы интерфейса с периферией (порт 61h). В свое время мы подробно разберемся с логическими операциями, сейчас следует лишь пояснить, что единица на выходе схемы логического умножения может появиться только в одном случае — когда на обоих входах единицы. Когда значение в регистре-счетчике становится равным нулю, на соответствующем входе схемы И формируется такая единица. И если при этом на втором входе, значение которого зависит от бита 0 порта 61h, также 1, то импульс от системных часов проходит на выход канала 2. Одновременно с пропуском импульса в канале 2 немедленно производится загрузка содержимого регистра-фиксатора (которое не изменилось, если его не изменили извне) в регистр-счетчик. Весь процесс с уменьшением содержимого регистра-счетчика повторяется заново. Теперь вы понимаете, что чем меньшее значение загружено в регистр-фиксатор, тем чаще будет происходить обнуление регистра-счетчика и тем чаще импульсы будут проходить на выход канала 2. А это означает большее значение *высоты* звука. Понятно, что максимальное значение частоты на входе 1 динамика — 1,19 МГц. Таким образом, импульс с выхода канала 2 попадает

на динамик, и если на последний подан ток, то возникает долгожданный звук. Подачей тока на динамик управляет бит 1 порта 61h. Как прервать звучание? Очевидно, для этого возможны два пути: первый — отключить ток, сбросив бит 1 порта 61h, второй — сбросить бит 0 порта 61h. Эти две возможности используют для создания различных звуковых эффектов. Сбрасывая и устанавливая эти биты, мы фактически определяем *длительность* звучания.

Если вы внимательно следили за всеми рассуждениями, то, наверное, без труда сможете понять, почему первый канал таймера формирует сигналы аппаратного прерывания от таймера 18,2 раза в секунду (на основании этих сигналов программы отслеживают время). Для этого BIOS во время загрузки после включения компьютера загружает в первый канал соответствующее значение.

Таким образом, наметились три последовательных действия, необходимые для программирования звукового канала таймера (они применимы и к остальным каналам).

1. Посредством порта 43h выбрать канал, задать режим работы и тип операции передачи значения в канал. В нашем случае соответствующее значение будет равно 10110110 = 0b6h.
2. Подать ток на динамик, установив бит 1 порта 61h.
3. Используя регистр AX, поместить нужное значение в порт 42h, определив тем самым нужную высоту тона.

Далее в листинге 7.1, приведена программа, реализующая некоторые звуки. Многие команды вам уже знакомы, некоторые мы пока еще не рассматривали, поэтому поясним их функциональное назначение. Подробно они будут рассмотрены в последующих главах. Для удобства в программе была использована макрокоманда delay, выполняющая задержку работы программы на заданное время. Подробнее механизм макроподстановок будет рассмотрен в главе 14. Сейчас только отметьте для себя, что введенная таким образом макрокоманда в тексте программы синтаксически ничем не отличается от других команд ассемблера, и это позволяет программисту при необходимости расширить стандартный набор команд ассемблера. Введите текст макрокоманды delay (строки 13-25) и воспринимайте ее чисто по функциональному назначению (задержка выполнения программы на промежуток времени, задаваемый значением ее операнда). Стоит отметить, что данная макрокоманда чувствительна к производительности процессора, из-за чего звуки на компьютерах с разными моделями процессоров могут не совпадать. Сегмент кода, как обычно, начинается с настройки сегментного регистра DS (строки 32-33) на начало сегмента данных. После этого строками 37-38 мы выполняем действия первого этапа — настройку канала 2, которая заключается в записи в регистр управления (порт 43h) байта состояния OB6h. На втором шаге мы должны установить биты 0 и 1 порта 61h. Предварительно необходимо извлечь содержимое этого порта. Это делается для того, чтобы выполнять установку битов 0 и 1, не изменяя содержимого остальных битов порта 61h (строки 39-41). Принцип формирования сигнала сирены заключается в том, что в цикле на единицу изменяется содержимое регистра-счетчика и делается небольшая задержка для того, чтобы сигнал некоторое время звучал с нужной высотой. Постепенное повышение, а за-

тем понижение высоты и дает нам эффект сирены. Строки 43-53 соответствуют циклу, в теле которого высота повышается, а строки 55-62 — циклу понижения тона. Оба цикла повторяются последовательно 5 раз. Контроль осуществляется с помощью переменной cnt, содержимое которой увеличивается на 1 в строке 69 и контролируется на равенство 5 в строках 71-72. Если cnt = 5, то команда CMP устанавливает определенные флаги. Последующая команда условного перехода JNE анализирует эти флаги и в зависимости от их состояния передает управление либо на метку, указанную в качестве операнда этой команды, либо на следующую за JNE команду. Цикл в программе ассемблера можно организовать несколькими способами; все они будут подробно рассмотрены в главах 10 и 11. В данном случае цикл организуется командой ШОР, которая в качестве операнда имеет имя метки. На эту метку и передается управление при выполнении команды ШОР. Но до того как передать управление, команда ШОР анализирует содержимое регистра ECX/CX, и если оно равно нулю, управление передается не на метку, а на следующую за LOOP команду. Если содержимое регистра ECX/CX не равно нулю, то оно уменьшается на единицу и управление передается на метку. Таким образом в ECX/CX хранится *счетчик цикла.* В нашей программе он загружается в ECX/CX в строках 42 и 54.

Листинг 7.1. Реализация сирены

```
<1> ;----Prg_7_1.asm--------
<2> ;Программа,имитирующая звук сирены.
<3> ;Изменение высоты звука от 450 до 2100 Гц.
<4> ;Используется макрос delay (задержка).
<5> ;При необходимости
<6> ;можно поменять значение задержки (по умолчанию - для процессора
        Pentium).
<7> ;------------------------
<8> masm
<9> model   small
<10>    stack   100h
<11>    ;макрос задержки, его текст ограничивается директивами macro и endm.
<12>    ;На входе - значение задержки (в мкс)
<13>    delay macro time
<14>        local   ext,iter
<15>        push    cx
<16>        mov     cx,time
<17>    ext:
<18>        push    cx
<19>        mov     cx,5000
<20>    iter:
<21>        loop    iter
<22>        pop     cx
<23>        loop    ext
<24>        Pop     cx
<25>    endm
<26>    .data                   ;сегмент данных
<27>    tonelow dw  2651        ;нижняя граница звучания 450 Гц
<28>    cnt     db  0           ;счетчик для выхода из программы
<29>    temp    dw  ?           ;верхняя граница звучания
<30>    .code                   ;сегмент кода
<31>    main:                   ;точка входа в программу
<32>        mov ax,@data        ;связываем регистр ds с сегментом
<33>        mov ds,ax           ;данных через регистр ax
<34>        mov ax,0            ;очищаем ax
<35>    go:
<36> ;заносим слово состояния 10110110b(0B6h) в командный регистр (порт 43h)
```

продолжение ↗

Листинг 7.1 *(продолжение)*

```
        raoval,0B6h
        out 43h,al
        in  al,61h          ;получим значение порта 61h в al
        or  al,3            ;инициализируем динамик и подаем ток в порт 61h
        out 61h,al
        mov cx,2083         ;количество шагов ступенчатого изменения тона
    musicup:
                            ;в ax значение нижней границы частоты
        mov ax,tonelow
        out 42h,al          ;в порт 42h младшее слово ax:al
        xchg al,ah          ;обмен между al и ah
        out 42h.al          ;в порт 42h старшее слово ax:ah
        add tonelow,1       ;повышаем тон
        delay 1             ;задержка на 1 мкс
        mov dx,tonelow      ;в dx текущее значение высоты
        mov temp,dx         ;temp - верхнее значение высоты
        loop musicup        ;повторить цикл повышения
        mov cx,2083         ;восстановить счетчик цикла
    musicdown:
        mov ax,temp         ;в ax верхнее значение высоты
        out 42h,al          ;в порт 42h младшее слово ax:al
        mov al,ah           ;обмен между al и ah
        out 42h,al          ;в порт 42h старшее слово ax:ah
        sub temp,1          ;понижаем высоту
        delay 1             ;задержка на 1 мкс
        loop musicdown      ;повторить цикл понижения
    nosound:
        in  al,61h          ;получим значение порта 61h в AL
        and al,0FCh         ;выключить динамик
        out 61h,al          ;в порт 61h
        mov dx,2651         ;для последующих циклов
        mov tonelow,dx
        inc cnt             ;увеличиваем счетчик проходов, то есть
                            ;количество звучаний сирены
        cmp cnt,5           ;5 раз ?
        jne go              ;если нет, идти на метку go
    exit:
        mov ax,4c00h;стандартный выход
        int 21h
    end main            ;конец программы
```

Среди файлов, прилагаемых к книге, в каталоге данной главы есть еще один пример использования команд ввода из порта и вывода в порт — это редактор CMOS-памяти. На данном этапе изучения ассемблера не стоит пытаться разобраться с ним, но впоследствии обязательно стоит к нему вернуться.

Работа с адресами и указателями

При написании программ на ассемблере производится интенсивная работа с адресами операндов, находящимися в памяти. Для поддержки такого рода операций есть специальная группа команд, в которую входят следующие команды.

- lea **<приемник>,<источник>** — загрузка *эффективного* адреса;
- lds **<приемник>,<источник>** — загрузка указателя в регистр сегмента данных ds;
- les **<приемник>,<источник>** — загрузка указателя в регистр дополнительного сегмента данных es;

- **lgs <приемник>,<источник>** — загрузка указателя в регистр дополнительного сегмента данных gs;
- **lfs <приемник>,<источник>** — загрузка указателя в регистр дополнительного сегмента данных fs;
- **lss <приемник>,<источник>** — загрузка указателя в регистр сегмента стека ss.

Команда LEA похожа на команду MOV тем, что она также производит пересылку, однако команда LEA производит пересылку не данных, а *эффективного адреса* данных (то есть смещения данных относительно начала сегмента данных) в регистр, указанный операндом **<приемник>**.

Часто для выполнения некоторых действий в программе недостаточно знать значение одного лишь эффективного адреса данных, а необходимо иметь полный указатель на данные. Вы помните, что полный указатель на данные состоит из сегментной составляющей и смещения. Все остальные команды этой группы позволяют получить в паре регистров такой полный указатель на операнд в памяти. При этом имя сегментного регистра, в который помещается сегментная составляющая адреса, определяется кодом операции. Соответственно, смещение помещается в регистр общего назначения, указанный операндом **<приемник>**. Но не все так просто с операндом <источник>. На самом деле в команде в качестве источника нельзя указывать непосредственно имя операнда в памяти, на который мы бы хотели получить указатель. Предварительно необходимо получить само значение полного указателя в некоторой области памяти и задать в команде получения полного адреса имя этой области. Для выполнения этого действия необходимо вспомнить директивы резервирования и инициализации памяти (см. главу 5). При применении этих директив возможен частный случай, когда в поле операндов указывается имя другой директивы определения данных (фактически, имя переменной). В этом случае в памяти формируется адрес этой переменной. Какой адрес будет сформирован (эффективный или полный), зависит от применяемой директивы. Если это DW, то в памяти формируется только 16-разрядное значение эффективного адреса, если же DD — в память записывается полный адрес. Размещение этого адреса в памяти следующее: в младшем слове находится смещение, в старшем — 16-разрядная сегментная составляющая адреса. Посмотрите на листинг 5.3 и рис. 5.20 (глава 5). В сегменте данных программы из листинга 5.3 переменные adr и adr_full иллюстрируют наш случай получения частичного и полного указателей на данные в памяти.

Например, при организации работы с цепочкой символов удобно поместить ее начальный адрес в некоторый регистр и далее в цикле модифицировать это значение для последовательного доступа к элементам цепочки. В листинге 7.2 производится копирование строки байтов str_1 в строку байтов str_2. В строках 13 и 14 в регистры SI и DI загружаются значения эффективных адресов переменных str_1 и str_2. В строках 18, 19 производится пересылка очередного байта из одной строки в другую. Указатели на позиции байтов в строках определяются содержимым регистров SI и DI. Для пересылки очередного байта необходимо увеличить на единицу регистры SI и DI, что и делается командами сложения INC (строки 20, 21). После этого программу необходимо зациклить до обработки всех символов строки.

Листинг **7.2.** Копирование строки

```
<1>   ; ——Prg_7_2.asm ———
<2> masm
<3> model small
<4> .data
<5> ...
<6> str_1 db "Ассемблер - базовый язык компьютера"
<7> str_2 db 35 dup (" ")
<8> full_pnt dd str_1
<9> ...
<10>    .code
<11>    start:
<12>    ...
<13>        lea si,str_1
<14>        lea di,str_2
<15>        les bx,full_pnt ;полный указатель на str1 в пару es:bx
<16>        mov cx,35        ;счетчик цикла для loop ml (глава 10)
<17>    ml:
<18>        mov al,[si]
<19>        mov [di],al
<20>        inc si
<21>        inc di
<22>    ;цикл на метку ml до пересылки всех символов (loop ml)
<23>    ...
<24>    end start
```

Необходимость использования команд получения полного указателя данных в памяти, то есть адреса сегмента и значения смещения внутри сегмента, возникает, в частности, при работе с цепочками. Мы рассмотрим этот вопрос в главе 12. В строке 8 листинга 7.2 в двойном слове full_pnt формируются сегментная часть адреса и смещение для переменной str_1. При этом два байта смещения занимает младшее слово fulLpnt, а значение сегментной составляющей адреса — старшее слово fulLpnt. В строке 15 командой LES эти компоненты адреса помещаются в регистры BX и ES.

Преобразование данных

К группе команд преобразования данных можно отнести множество команд процессора, но большинство из них имеют те или иные особенности, которые требуют отнести их к другим функциональным группам (см. рис 3.3). Поэтому из всей совокупности команд процессора непосредственно к командам преобразования данных можно отнести только одну команду

```
xlat [адрес_таблицы_перекодировки]
```

Это очень интересная и полезная команда. Ее действие заключается в том, что она замещает значение в регистре AL другим байтом из таблицы в памяти, расположенной по адресу, указанному операндом адрес_таблицы_перекодировки. Слово «таблица» весьма условно; по сути, это просто строка байтов. Адрес байта в строке, которым будет производиться замещение содержимого регистра AL, определяется суммой (BX) + (AL), то есть содержимое AL играет роль индекса в байтовом массиве.

При работе с командой XLAT обратите внимание на следующий тонкий момент. Хотя в команде указывается адрес строки байтов, из которой должно быть извлечено новое значение, этот адрес должен быть предварительно загружен (например, с помощью команды LEA) в регистр BX. Таким образом, операнд адрес_табли-

цы_перекодировки на самом деле не нужен (на это указывают квадратные скобки). Что касается строки байтов (таблицы перекодировки), то она представляет собой область памяти размером от 1 до 255 байт (диапазон числа без знака в 8-разрядном регистре).

В качестве иллюстрации работы данной команды мы рассмотрим программу из листинга 6.1 (см. главу 6). Вы помните, что эта программа преобразовывала двузначное шестнадцатеричное число, вводимое с клавиатуры (то есть в символьном виде), в эквивалентное двоичное представление в регистре AL. В листинге 7.3 приведен вариант этой программы с использованием команды XLAT.

Листинг 7.3. Использование таблицы перекодировки

```
<1> ;——Prg_7_3.asm————
<2> ;Программа преобразования двузначного шестнадцатеричного числа
<3> ;в двоичное представление с использованием команды xlat.
<4> ;Вход: исходное шестнадцатеричное число; вводится с клавиатуры.
<5> ;Выход: результат преобразования в регистре al
<6> masm
<7> model   small
<8> .data                   ;сегмент данных
<9> message db  "Введите две шестнадцатеричные цифры,$"
<10> tabl   db  48 dup(0),0,1,2,3,4,5,6,7,8,9, 7 dup (0)
<11>        db  0ah,0bh,0ch,0dh,0eh,0fh, 26 dup (0)
<12>        db  0ah,0bh,0ch,0dh,0eh,0fh, 152 dup (0)
<13>    .stack 256          ;сегмент стека
<14>    .code
<15>    ;начало сегмента кода
<16>    main proc         ;начало процедуры main
<17>        mov ax,@data;физический адрес сегмента данных в регистр ax
<18>        mov ds,ax    ;ax записываем в ds
<19>        lea bx,tabl ;загрузка адреса строки байт в регистр bx
<20>        mov ah,9
<21>        mov dx,offset message
<22>        int 21h       ;вывести приглашение к вводу
<23>        xor ax,ax     ;очистить регистр ax
<24>        mov ah,1h     ;значение lh в регистр ah
<25>        int 21h       ; вводим первую цифру в al
<26>        xlat          ; перекодировка первого введенного символа в al
<27>        mov dl,al
<28>        shl dl,4;сдвиг dl влево для освобождения места для младшей цифры
<29>        int 21h        ;ввод второго символа в al
<30>        xlat           ;перекодировка второго введенного символа в al
<31>        add al,dl      ;складываем для получения результата
<32>        mov ax,4c00h;пересылка 4c00h в регистр ax
<33>        int 21h         ;завершение программы
<34>    endp main          ;конец процедуры main
<36>    end main           ; конец программы с точкой входа main
```

Сама по себе программа проста; сложность вызывает обычно формирование таблицы перекодировки. Обсудим этот вопрос подробнее. Прежде всего нужно определиться со значениями тех байтов, которые вы будете изменять. В нашем случае это символы шестнадцатеричных цифр. В главе 6 мы рассматривали их ASCII-коды. Поэтому мы конструируем в сегменте данных таблицу, в которой на места байтов, соответствующих символам шестнадцатеричных цифр, помещаем их новые значения, то есть двоичные эквиваленты шестнадцатеричных цифр. Строки 10-12 листинга 7.3 демонстрируют, как это сделать. Байты этой таблицы, смещения которых не совпадают со значением кодов шестнадцатеричных цифр, нуле-

вые. Таковыми являются первые 48 байт таблицы, промежуточные байты и часть в конце таблицы. Желательно определить все 256 байт таблицы. Дело в том, что если мы ошибочно поместим в AL код символа, отличный от символа шестнадцатеричной цифры, то после выполнения команды XLAT получим непредсказуемый результат. В случае программы из листинга 7.3 это будет ноль, что не совсем корректно, так как непонятно, что же в действительности было в AL: код символа 0 или что-то другое. Поэтому, наверное, есть смысл здесь поставить «защиту от дурака», поместив в неиспользуемые байты таблицы какой-нибудь определенный символ. После каждого выполнения команды XLAT нужно будет просто контролировать значение в AL на предмет совпадения с этим символом, и если оно имело место, выдавать сообщение об ошибке.

После того как таблица составлена, с ней можно работать. В сегменте команд строка 19 инициализирует регистр BX значением адреса таблицы TABL. Далее все очень просто. Поочередно вводятся символы двух шестнадцатеричных цифр и производится их перекодировка в соответствующие двоичные эквиваленты. В остальном программа аналогична программе из листинга 6.1.

Для закрепления знаний и исследования трудных моментов запустите программу из листинга 7.3 под управлением отладчика.

Работа со стеком

Стек — это область памяти, специально выделяемая для временного хранения данных программы. Важность стека определяется тем, что для него в структуре программы предусмотрен отдельный сегмент. На тот случай, если программист забыл описать сегмент стека в своей программе, компоновщик tlink выдаст предупреждающее сообщение.

Для работы со стеком предназначены три регистра:

- SS — регистр сегмента стека;
- SP/ESP — регистр указателя стека;
- BP/EBP — регистр указателя базы кадра стека.

Размер стека зависит от режима работы процессора и ограничивается значением 64 Кбайт (или 4 Гбайт в защищенном режиме). В каждый момент времени доступен только один стек, адрес сегмента которого содержится в регистре SS. Этот стек называется *текущим*. Для того чтобы обратиться к другому стеку («переключить стек»), необходимо загрузить в регистр SS другой адрес. Регистр SS автоматически используется процессором для выполнения всех команд, работающих со стеком.

Перечислим еще некоторые особенности работы со стеком.

- Запись и чтение данных в стеке осуществляются в соответствии с принципом LIFO (Last In First Out — «последним пришел, первым ушел»).
- По мере записи данных в стек последний растет в сторону младших адресов. Эта особенность заложена в алгоритм команд работы со стеком.
- При использовании регистров ESP/SP и EBP/BP для адресации памяти ассемблер автоматически считает, что содержащиеся в нем значения представляют собой смещения относительно сегментного регистра SS.

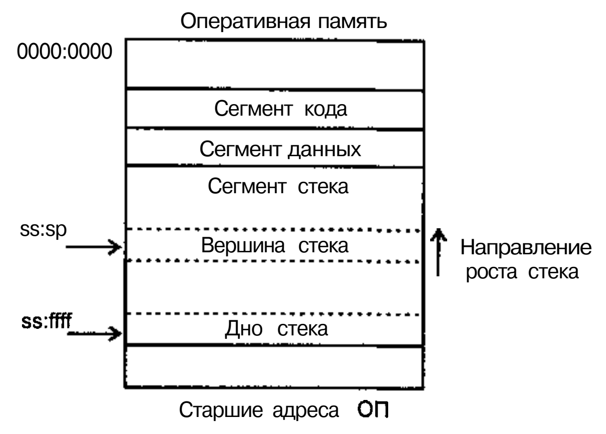


Рис. 7.3. Концептуальная схема организации стека

В общем случае стек организован так, как показано на рис. 7.3.

Регистры SS, ESP/SP и EBP/BP[1] используются комплексно, и каждый из них имеет свое функциональное назначение. Регистр ESP/SP всегда указывает на вершину стека, то есть содержит смещение, по которому в стек был занесен последний элемент. Команды работы со стеком неявно изменяют этот регистр так, чтобы он указывал всегда на последний записанный в стек элемент. Если стек пуст, то значение ESP равно адресу последнего байта сегмента, выделенного под стек. При занесении элемента в стек процессор уменьшает значение регистра ESP, а затем записывает элемент по адресу новой вершины. При извлечении данных из стека процессор копирует элемент, расположенный по адресу вершины, а затем увеличивает значение регистра указателя стека ESP. Таким образом, получается, что стек растет вниз, в сторону уменьшения адресов.

Что нужно сделать для получения доступа к элементам не на вершине, а внутри стека? Для этого применяют регистр EBP. Регистр EBP — регистр указателя базы кадра стека. Например, типичным приемом при входе в подпрограмму является передача нужных параметров путем записи их в стек. Если подпрограмма тоже активно работает со стеком, то доступ к этим параметрам становится проблематичным. Выход в том, чтобы после записи нужных данных в стек сохранить адрес вершины стека в указателе базы кадра стека — регистре EBP. Значение в EBP в дальнейшем можно использовать для доступа к переданным параметрам.

Начало стека расположено в старших адресах памяти. На рис. 7.3 этот адрес обозначен парой SS:ffff. Смещение ffff приведено здесь условно. Реально это значение определяется величиной, которую программист задает при описании сегмента стека в своей программе. К примеру, для программы из листинга 7.1 началу стека будет соответствовать пара SS:0100h. Адресная пара SS:ffff — это максимальное для реального режима значение адреса начала стека, так как размер сегмента в нем ограничен величиной 64 Кбайт (0ffffh).

1 Какой из регистров применяется для адресации, 32-разрядный или 16-разрядный, зависит от значения модификатора use16 или use32 в директиве сегментации segment (см. главу 5). При наличии директив упрощенного описания сегментов модификаторы use16 или use32 используются в директиве model.

Для организации работы со стеком существуют специальные команды записи и чтения.

Команда PUSH выполняет запись значения **<источник>** в вершину стека:

```
push <источник>
```

Интерес представляет алгоритм работы этой команды, который включает два действия (рис. 7.4):

1. Значение SP уменьшается на 2:

   (SP) = (SP) - 2

2. Значение источника записывается по адресу, указываемому парой SS:SP.

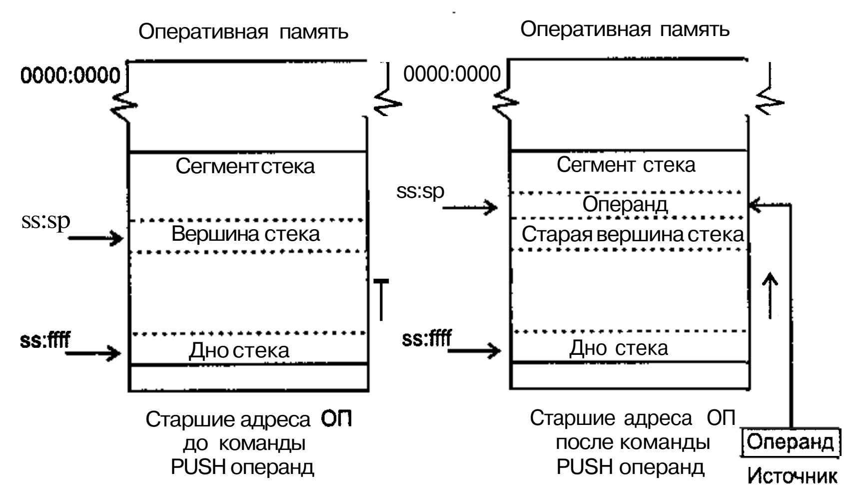


Рис. 7.4. Принцип работы команды PUSH

Команда POP выполняет запись значения из вершины стека по месту, указанному операндом **<приемник>** (значение при этом «снимается» с вершины стека):

```
pop <приемник>
```

Алгоритм работы команды POP обратен алгоритму команды PUSH (рис. 7.5).

1. Запись содержимого вершины стека по месту, указанному операндом **<приемник>**.
2. Увеличение значения SP:

   (SP) = (SP) + 2

Команда PUSHA предназначена для групповой записи в стек. По этой команде в стек последовательно записывается содержимое регистров AX, CX, DX, BX, SP, BP, SI, DI. **Заметим,** что записывается оригинальное содержимое SP, то есть то, которое было до выдачи команды PUSHA (рис. 7.6).

Команда PUSHAW почти идентична команде PUSHA. В чем разница? В главе 5 мы обсуждали один из атрибутов сегмента — *атрибут размера сегмента.* Он может принимать **значения use16** или **use32**:

- **use16** — алгоритм работы PUSHAW аналогичен алгоритму PUSHA;
- use32 — алгоритм работы команды PUSHAW не меняется (то есть она нечувствительна к разрядности сегмента и всегда работает с регистрами размером в слово — AX, CX, DX, BX, SP, BP, SI, DI), а команда PUSHA чувствительна к разрядности

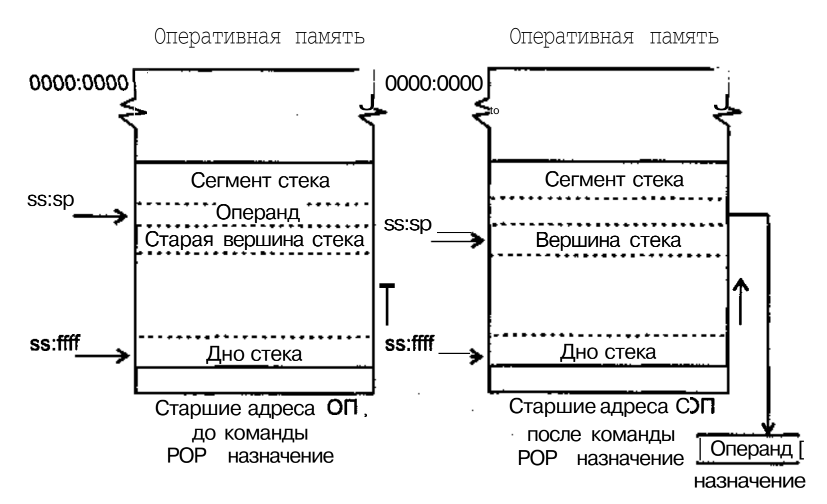


Рис. 7.5. Принцип работы команды POP

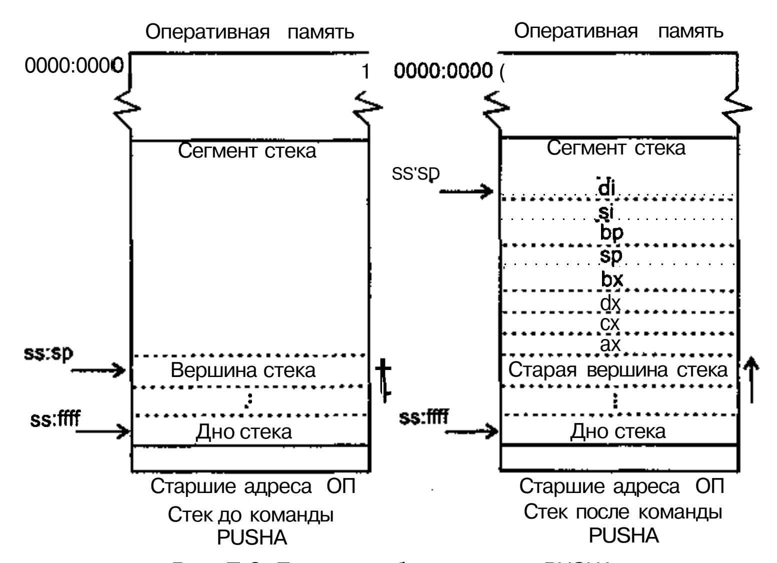


Рис. 7.6. Принцип работы команды PUSHA

сегмента и при указании 32-разрядного сегмента работает с соответствующими 32-разрядными регистрами (то есть EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI).

Команда PUSHAD — выполняется аналогично команде PUSHA, но есть некоторые особенности, которые вы можете найти в приложении.

Следующие три команды выполняют действия, обратные действиям описанных ранее команд:

- POPA;
- POPAW;
- POPAD.

Представленная далее группа команд позволяет сохранить в стеке регистр флагов и записать слово или двойное слово. Отметим, что перечисленные команды — единственные в системе команд процессора, которые позволяют получить доступ (и которые нуждаются в этом доступе) ко всему содержимому регистра флагов.

Команда PUSHF сохраняет регистр флагов в стеке. Работа этой команды зависит от атрибута размера сегмента:

- use16 — в стек записывается регистр FLAGS размером два байта;
- use32 — в стек записывается регистр EFLAGS размером четыре байта.

Команда PUSHFW сохраняет в стеке регистр флагов размером в слово. С атрибутом use16 всегда работает так же, как команда PUSHF.

Команда PUSHFD сохраняет в стеке регистр флагов FLAGS или EFLAGS в зависимости от атрибута размера сегмента (то есть то же, что и PUSHF).

Следующие три команды также выполняют действия, обратные действиям рассмотренных выше команд:

- POPF;
- POPFW;
- POPFD.

Работать со стеком приходится постоянно, поэтому к этому вопросу мы будем возвращаться еще не раз. Отметим основные виды операций, когда использование стека практически неизбежно:

- вызов подпрограмм;
- временное сохранение значений регистров;
- определение локальных переменных в процедуре.

Итоги

- Основная команда пересылки данных — MOV. Операнды этой команды (как и большинства других команд, берущих значения из памяти) должны быть согласованы по разрядности. Хотя обычно действуют правила умолчания, в сомнительных ситуациях лучше явно указывать разрядность операндов с помощью оператора PTR.
- Управление периферией компьютера в общем случае осуществляется с использованием всего двух команд ввода-вывода — IN и OUT.
- В процессе работы программы динамически можно получить как эффективный, так и полный (физический) адрес памяти. Для этого язык ассемблера предоставляет группу команд получения указателей памяти.
- Архитектура процессора предоставляет в распоряжение программиста специфическую, но весьма эффективную структуру — стек. Система команд поддерживает все необходимые операции со стеком. Подробнее со стеком мы познакомимся при изучении модульного программирования на ассемблере.

Глава 8
Арифметические команды

- **Форматы арифметических данных**
- **Арифметические операции над двоичными числами**
- **Арифметические операции над десятичными (BCD) числами**

Одной из причин, постоянно заставляющих человека совершенствовать средства для выполнения вычислений, — желание эффективно, быстро и без ошибок решать различные счетные задачи. Для начала мечтой людей была автоматизация выполнения простейших арифметических действий. Первая реализованная попытка — начало XVII в., 1623 г. Ученый В. Шикард создает машину, умеющую складывать и вычитать числа. Знаменитый французский ученый и философ Блез Паскаль в 1642 г. изобрел первый *арифмометр,* основным элементом в котором было зубчатое колесо. Изобретение этого колеса уже само по себе было ключевым событием в истории вычислительной техники, подобно лампам и транзисторам в наше время. Правнуки этого колеса еще совсем недавно, каких-нибудь два-три десятка лет назад, использовались в арифмометрах (соответствующая модель была создана в 1842 г.) на столах советских бухгалтеров. Тот, кому довелось поработать на этих арифмометрах, вряд ли вспомнят о высокой эффективности вычислительного процесса — слишком велика была зависимость от человеческого фактора. Снизить эту зависимость удалось лишь в середине прошлого века, когда появились первые ЭВМ на лампах, потом на транзисторах и, наконец, на микросхемах различной интеграции. Таким образом, путь к эффективному автоматизированному решению для проведения расчетов растянулся почти на три столетия. Тем не менее, именно благодаря стремлению разгрузить голову от рутины человек имеет сегодня определенные достижения в области компьютерной техники.

Любой компьютер, от самого примитивного до супермощного, имеет в своей системе команд команды для выполнения арифметических действий. Работая

с компьютером при помощи языков высокого уровня, мы воспринимаем возможность проведения расчетных действий как нечто должное, забывая при этом, что компилятор даже очень развитого языка программирования превращает все самые высокоуровневые действия в унылую последовательность машинных команд. Конечно, мало кому придет в голову писать серьезную расчетную задачу на ассемблере. Но даже в системных программах часто требуется проведение небольших вычислений. Поэтому важно разобраться с этой группой команд. К тому же она, на удивление, очень компактна и не избыточна.

Процессор может выполнять целочисленные операции и операции с плавающей точкой. Для этого в его архитектуре есть два отдельных устройства, каждое из которых имеет свою систему команд. В принципе, целочисленное устройство может взять на себя многие функции устройства с плавающей точкой, но это потребует больших вычислительных затрат. Устройство с плавающей точкой и его система команд будут рассмотрены в главе 17. Для большинства задач, использующих язык ассемблера, достаточно целочисленной арифметики.

Обзор

Целочисленное вычислительное устройство поддерживает чуть больше десятка арифметических команд. На рис. 8.1 приведена классификация команд этой группы.

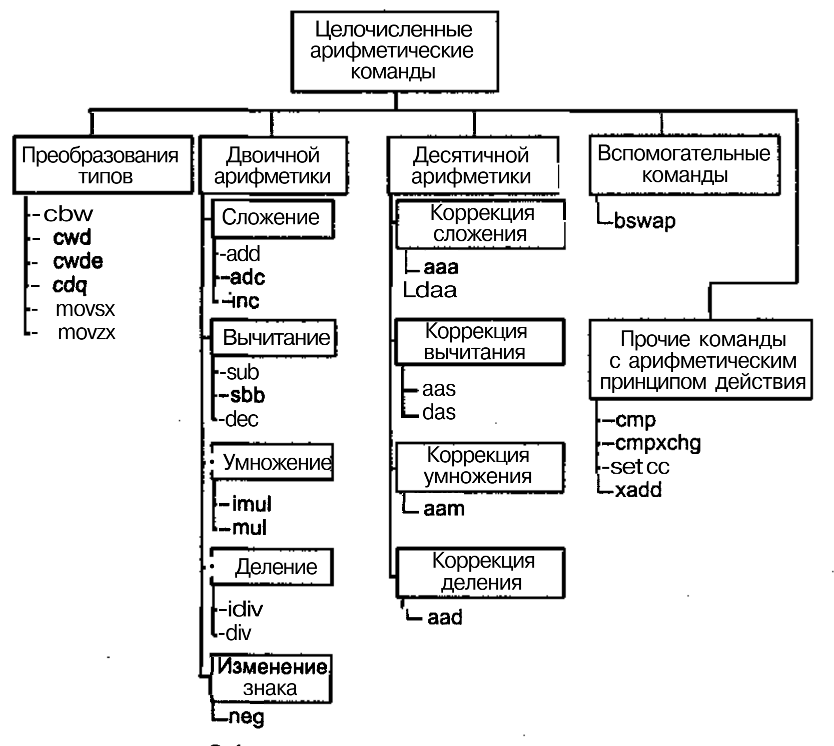


Рис. 8.1. Классификация арифметических команд

Группа арифметических целочисленных команд работает с двумя типами чисел:

* целыми двоичными числами, которые могут иметь или не иметь знаковый разряд, то есть быть числами со знаком или без знака;
* целыми десятичными числами.

В главе 3 мы обсуждали вопрос о форматах данных, поддерживаемых архитектурой IA-32. Рассмотрим теперь форматы данных, с которыми работают арифметические команды.

Целые двоичные числа

Целое двоичное число — это число, закодированное в двоичной системе счисления. В архитектуре IA-32 размерность целого двоичного числа может составлять 8, 16 или 32 бита. Знак двоичного числа определяется тем, как интерпретируется старший бит в представлении числа. Это 7-й, 15-й или 31-й биты для чисел соответствующей размерности (см. главу 5). При этом интересно то, что среди арифметических команд есть всего две, которые действительно учитывают этот старший разряд как знаковый, — это команды целочисленного умножения IMUL и деления IDIV. В остальных случаях ответственность за действия со знаковыми числами и, соответственно, со знаковым разрядом ложится на программиста. К этому вопросу мы вернемся чуть позже. Диапазон значений двоичного числа зависит от его размера и трактовки старшего бита либо как старшего значащего бита числа, либо как бита знака числа (табл. 8.1).

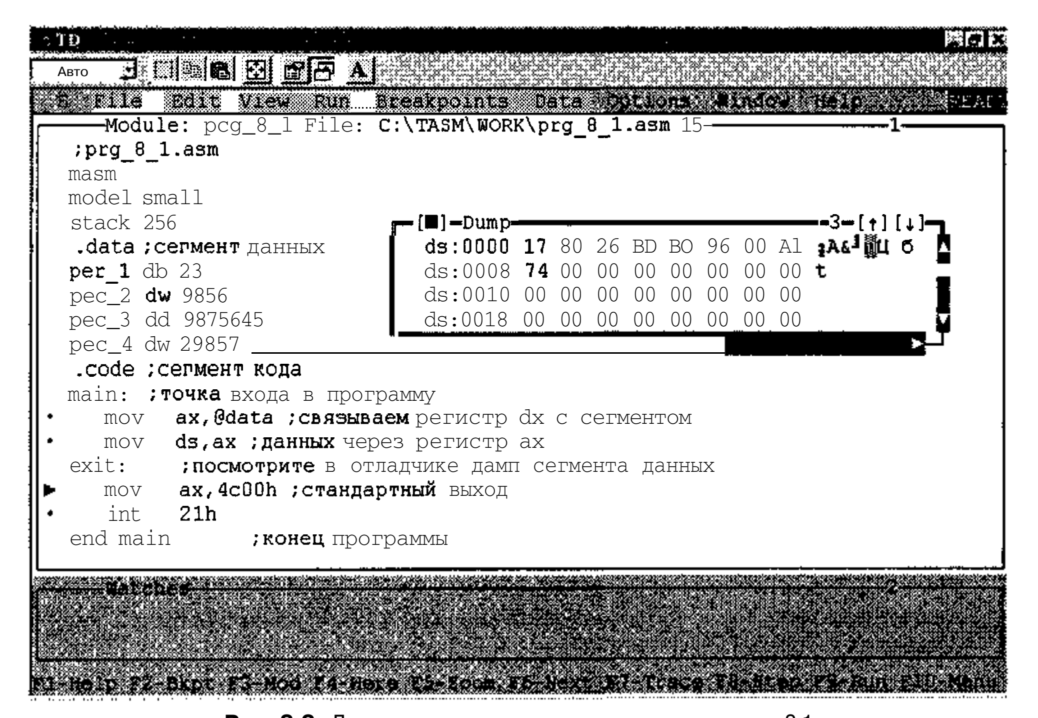


Рис. 8.2. Дамп памяти для сегмента данных листинга 8.1

Таблица 8.1. Диапазон значений двоичных чисел

Размерность поля	Целое без знака	Целое со знаком
Байт	0...255	-128...+ 127
Слово	0...65 535	-32 768...+32 767
Двойное слово	0...4 294 967 295	-2 147 483 648...+2 147 483 647

Как описать целые двоичные числа в программе? Это делается с использованием директив описания данных DB, DW и DD. В главе 5 описаны возможные варианты содержимого полей операндов этих директив и диапазоны их значений. К примеру, последовательность описаний двоичных чисел из сегмента данных листинга 8.1 (помните о принципе «младший байт по младшему адресу») будет выглядеть в памяти так, как показано на рис. 8.2.

Листинг 8.1. Числа с фиксированной точкой

```
;prg_8_1.asm
masm
model   small
stack   256
.data                 ;сегмент данных
per_1   db  23
per_2   dw  9856
per_3   dd  9875645
per_4   dw  29857
.code                 ;сегмент кода
main:                 ;точка входа в программу
    mov ax,@data;связываем регистр dx с сегментом
    mov ds,ax         ;данных через регистр ax
exit:                 ;посмотрите в отладчике дамп сегмента данных
    mov ax,4c00h;стандартный выход
    int 21h
end main              ;конец программы
```

Десятичные числа

Десятичные числа — специальный вид представления числовой информации, в основу которого положен принцип кодирования каждой десятичной цифры числа группой из четырех битов. При этом каждый байт числа содержит одну или две десятичные цифры в так называемом *двоично-десятичном коде* (Binary-Coded Decimal, BCD). Процессор хранит BCD-числа в двух форматах (рис. 8.3).

- В *упакованном формате* каждый байт содержит две десятичные цифры. Десятичная цифра представляет собой двоичное значение в диапазоне от 0 до 9 размером четыре бита. При этом код старшей цифры числа занимает старшие четыре бита. Следовательно, диапазон представления десятичного упакованного числа в одном байте составляет от 00 до 99.
- В *неупакованном формате* каждый байт содержит одну десятичную цифру в четырех младших битах. Старшие четыре бита имеют нулевое значение. Это так называемая *зона*. Следовательно, диапазон представления десятичного неупакованного числа в одном байте составляет от 0 до 9.

Как описать двоично-десятичные числа в программе? Для этого можно использовать только две директивы описания и инициализации данных — DB и DT. Воз-

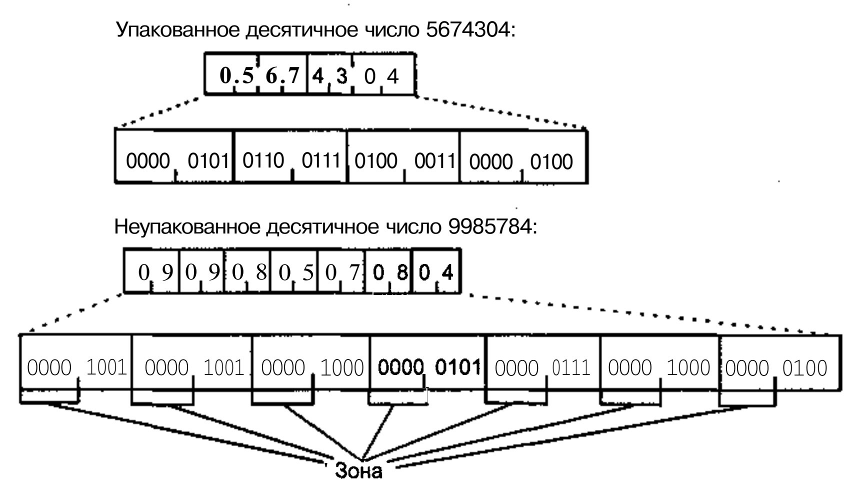


Рис. 8.3. Представление BCD-чисел

можность применения только этих директив для описания BCD-чисел обусловлена тем, что к таким числам также применим принцип «младший байт по младшему адресу», что, как мы увидим далее, очень удобно для их обработки. И вообще, при использовании такого типа данных, как BCD-числа, порядок описания этих чисел в программе и алгоритм их обработки — это дело вкуса и личных пристрастий программиста, что станет более ясным после того, как мы далее рассмотрим основы

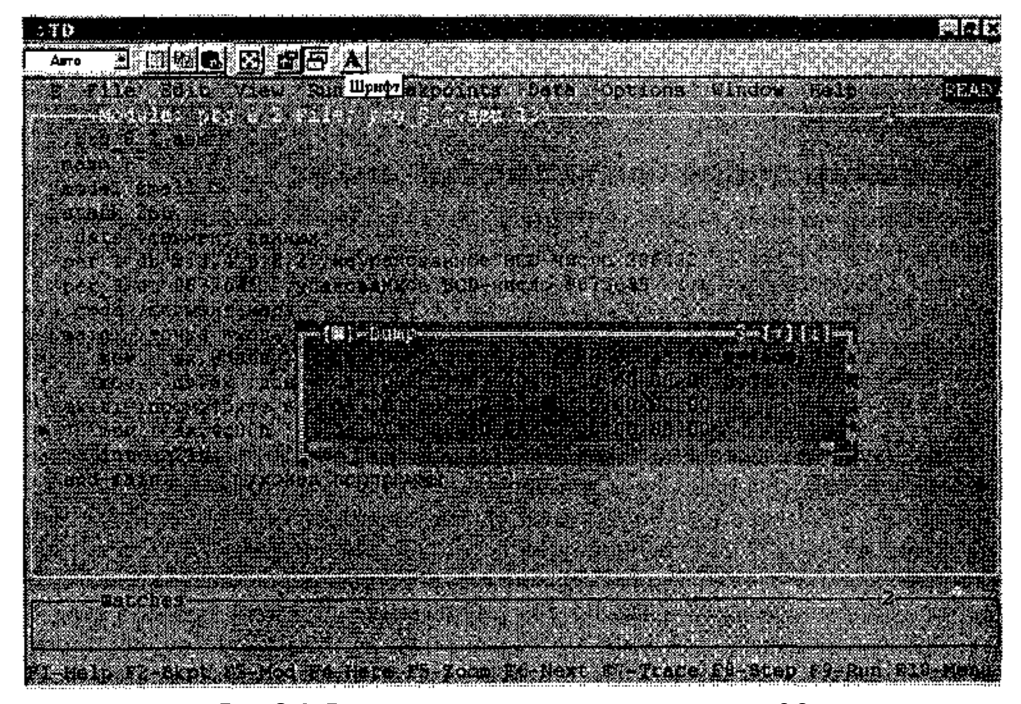


Рис. 8.4. Дамп памяти для сегмента данных листинга 8.2

работы с BCD-числами. К примеру, приведенная в сегменте данных листинга 8.2 последовательность описаний BCD-чисел будет выглядеть в памяти так, как показано на рис. 8.4.

Листинг 8.2. BCD-числа

```
;prg_8_2.asm
masm
model   small
stack   256
.data                       ;сегмент данных
per_l   db  2,3,4,6,8,2 ;неупакованное BCD-число 286432
per_3   dt  9875645      ;упакованное BCD-число 9875645
.code                       ;сегмент кода
main:                      ;точка входа в программу
    mov ax,@data            ;связываем регистр dx с сегментом
    mov ds,ax               ;данных через регистр ax
exit:                       ;посмотрите в отладчике дамп сегмента данных
    mov ax,4c00h            ;стандартный выход
    int 21h
end main                    ;конец программы
```

После столь подробного обсуждения форматов данных, с которыми работают арифметические операции, можно приступить к рассмотрению средств их обработки на уровне системы команд процессора.

Арифметические операции над целыми двоичными числами

В данном разделе мы рассмотрим особенности каждого из четырех основных арифметических действий для целых двоичных чисел со знаком и без знака. Дополнительные сведения о возможностях ассемблера по обработке целых двоичных чисел можно получить в [8].

Сложение двоичных чисел без знака

Процессор выполняет сложение операндов по правилам сложения двоичных чисел. Проблем не возникает до тех пор, пока значение результата не превышает размерности поля операнда (см. табл. 8.1). Например, при сложении операндов размером в байт результат не должен превышать 255. Если это происходит, то результат оказывается неверен. Рассмотрим, почему. К примеру, выполним сложение: 254 + + 5 = 259 в двоичном виде:

$$11111110 + 0000101 - 1\ 00000011.$$

Результат вышел за пределы восьми битов, и правильное его значение укладывается в 9 битов, а в 8-разрядном поле операнда осталось значение 3, что, конечно, неверно. В процессоре подобный исход сложения прогнозируется, и предусмотрены специальные средства для фиксации подобных ситуаций и их обработки. Так, для фиксации ситуации выхода за разрядную сетку результата, как в данном случае, предназначен *флаг переноса* CF. Он располагается в бите 0 регистра флагов EFLAGS/FLAGS. Именно установкой этого флага фиксируется факт переноса единицы из старшего разряда операнда. Естественно, что программист должен предусматривать возможность такого исхода операции сложения и средства для кор-

ректировки. Это предполагает включение фрагментов кода после операции сложения, в которых анализируется состояние флага CF. Этот анализ можно провести различными способами. Самый простой и доступный — использовать команду условного перехода JC. Эта команда в качестве операнда имеет имя метки в текущем сегменте кода. Переход на метку осуществляется в случае, если в результате работы предыдущей команды флаг CF установился в 1. Команды условных переходов будут рассматриваться в главе 10.

Если теперь посмотреть на рис. 8.1, то видно, что в системе команд процессора имеются три команды двоичного сложения:

- команда *инкремента,* то есть увеличения значения операнда на 1:

  ```
  inc операнд
  ```

- команда сложения (**операнд_1 = операнд_1 + операнд_2**):

  ```
  add операнд_1,операнд_2
  ```

- команда сложения с учетом флага переноса CF (**операнд_1 = операнд_1 + операнд_2 + значение_CF**):

  ```
  adc операнд_1,операнд_2
  ```

Обратите внимание на последнюю команду — это команда сложения, учитывающая перенос единицы из старшего разряда. Механизм появления такой единицы мы уже рассмотрели. Таким образом, команда ADC является средством процессора для сложения длинных двоичных чисел, размерность которых превосходит поддерживаемые процессором размеры стандартных полей.

Рассмотрим пример вычисления суммы чисел (листинг 8.3).

Листинг 8.3. Вычисление суммы чисел

```
<1> ;prg_8_3.asm
<2> masm
<3> model   small
<4> stack   256
<5> .data
<6> a   db  254
<7> .code                   ;сегмент кода
<8> main:
<9>     mov ax,@data
<10>        mov ds,ax
<11>    ;...
<12>        xor ax,ax
<13>        add al,17
<14>        add al,a
<15>        jnc ml      ;если нет переноса, то перейти на ml
<16>        adc ah,0    ;в ах сумма с учетом переноса
<17>   ml: ;...
<18>        exit:
<19>        mov ax,4c00h;стандартный выход
<20>        int 21h
<21>    end main        ;конец программы
```

В строках 13-14 создана ситуация, когда результат сложения выходит за границы операнда. Эта возможность учитывается строкой 15, где команда JNC (хотя можно было обойтись и без нее) проверяет состояние флага CF. Если он установлен в 1, то результат операции получился большим по размеру, чем операнд, и для его корректировки необходимо выполнить некоторые действия. В данном случае мы просто полагаем, что границы операнда расширяются до размера AX, для чего

учитываем перенос в старший разряд командой ADC (строка 16). Напомню, что исследовать работу команд сложения без учета знака вы можете в отладчике. Для этого введите в текстовом редакторе текст листинга 8.3, получите исполняемый модуль, запустите отладчик и откройте в нем окна командами View ▸ Dump и View ▸ Registers. Далее, в пошаговом режиме отладки можно более наглядно проследить за всеми процессами, происходящими в процессоре во время работы программы.

Сложение двоичных чисел со знаком

Теперь настала пора раскрыть небольшой секрет. Дело в том, что на самом деле процессор не подозревает о различии между числами со знаком и числами без знака. Вместо этого у него есть средства фиксации возникновения характерных ситуаций, складывающихся в процессе вычислений. Некоторые из них мы рассмотрели ранее при обсуждении команд сложения чисел без знака — это учет флага переноса CF. Установка этого флага в 1 говорит о том, что произошел выход за пределы разрядности операндов. Далее с помощью команды ADC можно учесть возможность такого выхода (переноса из младшего разряда) во время работы программы. Другое средство фиксации характерных ситуаций в процессе арифметических вычислений — регистрация состояния старшего (знакового) разряда операнда, которое осуществляется с помощью флага переполнения OF в регистре EFLAGS (бит И).

В главе 4 мы рассматривали, как представляются числа в компьютере. При этом отмечали, что положительные числа представляются в двоичном коде, а отрицательные — в дополнительном. Рассмотрим различные варианты сложения чисел. Примеры призваны показать поведение двух старших битов операндов и правильность результата операции сложения.

Первый вариант сложения чисел:

30566 = 01110111 01100110

+

00687 = 00000010 10101111

=

31253=0111101000010101.

Следим за переносами из 14-го и 15-го разрядов и за правильностью результата: переносов нет, результат правильный.

Второй вариант сложения чисел:

30566 = 01110111 01100110

+

30566 = 01110111 01100110

=

61132 = 11101110 11001100.

Произошел перенос из 14-го разряда; из 15-го разряда переноса нет. Результат неправильный, так как имеется *переполнение* — значение числа получилось больше, чем то, которое может иметь 16-разрядное число со знаком (+32 767).

Третий вариант сложения чисел:

-30566 = 10001000 10011010
+
-04875 = 11101100 11110101
=
-35441 = 01110101 10001111.

Произошел перенос из 15-го разряда, из 14-го разряда нет переноса. Результат неправильный, так как вместо отрицательного числа получилось положительное (в старшем бите находится 0).

Четвертый вариант сложения чисел:

-4875 = 11101100 11110101
+
-4875 = 11101100 11110101
=
-9750 = 11011001 11101010.

Есть переносы из 14-го и 15-го разрядов. Результат правильный.

Таким образом, мы исследовали все случаи и выяснили, что ситуация *переполнения* (установка флага OF в 1) происходит при переносе:

* из 14-го разряда (для положительных чисел со знаком);
* из 15-го разряда (для отрицательных чисел).

И, наоборот, переполнения не происходит (то есть флаг OF сбрасывается в 0), если есть перенос из обоих разрядов или перенос отсутствует в обоих разрядах.

Таким образом, ситуация переполнение регистрируется процессором с помощью флага переполнения OF. Дополнительно к флагу OF при переносе из старшего разряда устанавливается в 1 и флаг переноса CF. Так как процессор не знает о существовании чисел со знаком и без знака, то вся ответственность за правильность действий с получившимися числами ложится на программиста. Теперь, наверное, понятно, почему мы столько внимания уделили тонкостям сложения чисел со знаком. Учтя все это, мы сможем организовать правильный процесс сложения чисел — будем анализировать флаги CF и OF и принимать правильное решение! Проанализировать флаги CF и OF можно командами условного перехода JC\JNC и JO\JNO соответственно (глава 10).

Что же касается команд сложения чисел со знаком, то из изложенного ранее понятно, что в архитектуре IA-32 сами команды сложения чисел со знаком те же, что и для чисел без знака.

Вычитание двоичных чисел без знака

Аналогично анализу операции сложения, порассуждаем над сутью процессов, происходящих при выполнении вычитания.

- Если уменьшаемое больше вычитаемого, то проблем нет, — разность положительна, результат верен.
- Если уменьшаемое меньше вычитаемого, возникает проблема: результат меньше 0, а это уже число со знаком. В этом случае результат необходимо *завернуть*. Что это означает? При обычном вычитании (в столбик) делают заем 1 из старшего разряда. Процессор поступает аналогично, то есть занимает 1 из разряда, следующего за старшим в разрядной сетке операнда. Поясним на примере.

Первый вариант вычитания чисел:

05 = 0000000000000101
-10 = 0000000000001010.

Для того чтобы произвести вычитание, произведем воображаемый заем из старшего разряда:

100000000 00000101
–
0000000000001010
=
11111111 11111011.

Тем самым, по сути, выполняется действие (65 536 + 5) - 10 = 65 531, 0 здесь как бы эквивалентен числу 65 536. Результат, конечно, неверен, но процессор считает, что все нормально, тем не менее, факт заема единицы он фиксирует, устанавливая флаг переноса CF. Посмотрите еще раз внимательно на результат операции вычитания. Это же число -5 в дополнительном коде! Проведем эксперимент: представим разность в виде суммы 5 + (-10).

Второй вариант вычитания чисел:

5 = 00000000 00000101
+
(-10) = 11111111 11110110
=
11111111 11111011.

То есть мы получили тот же результат, что и в предыдущем примере. Таким образом, после команды вычитания чисел без знака нужно анализировать состояние флага CF. Если он установлен в 1, это говорит о том, что произошел заем из старшего разряда и результат получился в дополнительном коде.

Аналогично командам сложения группа команд вычитания состоит из минимально возможного набора. Эти команды выполняют вычитание по алгоритмам, которые мы сейчас рассматриваем, а учет особых ситуаций должен производиться самим программистом.

- Команда декремента выполняет уменьшения значения операнда на 1:

  `dec операнд`

- Команда вычитания (операнд_1 = операнд_1 - операнд_2):

  `sub операнд_1,операнд_2`

- Команда вычитания с учетом заема, то есть флага CF (операнд_1 = операнд_1 - операнд_2 - значение_CF):

  `sbb операнд_1,операнд_2`

Как видите, среди команд вычитания есть команда SBB, учитывающая флаг переноса CF. Эта команда подобна ADC, но теперь уже флаг CF играет роль индикатора заема 1 из старшего разряда при вычитании чисел.

Рассмотрим пример (листинг 8.4) программной обработки ситуации, рассмотренной ранее для второго варианта вычитания чисел.

Листинг 8.4. Проверка при вычитании чисел без знака

```
<1> ;prg_8_4.asm
<2> masm
<3> model   small
<4> stack   256
<5> .data
<6> .code                       ;сегмент кода
<7> main:                       ;точка входа в программу
<8> ;...
<9>     xor ax,ax
<10>        mov al,5
<11>        sub al,10
<12>        jnc ml              ;нет  переноса?
<13>        neg al              ;в al модуль результата
<14>    ml: ;...
<15>        exit:
<16>        mov ax,4c00h;стандартный выход
<17>        int 21h
<18>    end main            ;конец программы
```

В этом примере в строке 1 1 выполняется вычитание. С указанными для этой команды вычитания исходными данными результат получается в дополнительном коде (отрицательный). Для того чтобы преобразовать результат к нормальному виду (получить его модуль), применяется команда NEG, с помощью которой получается дополнение операнда. В нашем случае мы получили дополнение дополнения, или модуль отрицательного результата. А тот факт, что это на самом деле число отрицательное, отражен в состоянии флага CF. Дальше все зависит от алгоритма обработки. Исследуйте программу в отладчике.

Вычитание двоичных чисел со знаком

Вычитание двоичных чисел со знаком выполнять несколько сложнее. Последний пример показал то, что процессору незачем иметь два устройства — сложения и вычитания. Достаточно наличия только одного — устройства сложения. Но для вычитания способом сложения чисел со знаком оба операнда (и уменьшаемое, и вычитаемое) необходимо представлять в дополнительном коде. Результат тоже нужно рассматривать как значение в дополнительном коде. Но здесь возникают сложности. Прежде всего, они связаны с тем, что старший бит операнда рассматривается как знаковый. Рассмотрим пример вычитания 45 - (-127).

Первый вариант вычитания чисел со знаком:

45 =00101101

–

-127 = 10000001

=

-44 = 1010 1100.

Судя по знаковому разряду, результат получился отрицательный, что, в свою очередь, говорит о том, что число нужно рассматривать как дополнение, равное -44. Правильный результат должен быть равен 172. Здесь мы, как и в случае знакового сложения, встретились с *переполнением мантиссы,* когда значащий разряд числа изменил знаковый разряд операнда. Отследить такую ситуацию можно по содержимому флага переполнения OF. Его установка в 1 говорит о том, что резуль-

тат вышел за диапазон представления знаковых чисел (то есть изменился старший бит) для операнда данного размера и программист должен предусмотреть действия по корректировке результата.

Следующее вычитание чисел со знаком выполним способом сложения:

-45 - 45 - -45 + (−45)= -90.

-415 = 1101 0011

+

-45 =11010011

-90 = 1010 0110.

Здесь все нормально, флаг переполнения OF сброшен в 0, а 1 в знаковом разряде говорит о том, что значение результата — число в дополнительном коде.

Вычитание и сложение операндов большой размерности

Если вы заметили, команды сложения и вычитания работают с операндами фиксированной размерности: 8, 16, 32 бита. А что делать, если нужно сложить числа большей размерности, например 48 битов, используя 16-разрядные операнды? К примеру, сложим два 48-разрядных числа (рис. 8.5).

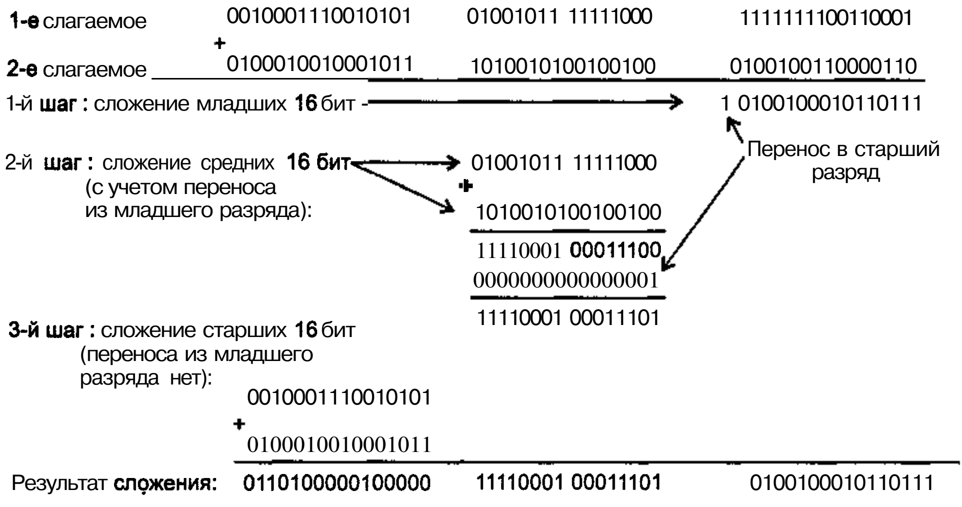


Рис. 8.5. Сложение операндов большой размерности

Рисунок по шагам иллюстрирует технологию сложения длинных чисел. Видно, что процесс сложения многобайтных чисел происходит так же, как и при сложении двух чисел «в столбик», — с осуществлением при необходимости переноса 1 в старший разряд. Если нам удастся запрограммировать этот процесс, то мы значительно расширим диапазон двоичных чисел, над которыми можно выполнять операции сложения и вычитания.

Принцип вычитания чисел с диапазоном представления, превышающим стандартные разрядные сетки операндов, тот же, что и при сложении, то есть использу-

ется флаг переноса CF. Нужно только представлять себе процесс вычитания в столбик и правильно комбинировать команды процессора с командой SBB. Чтобы написать достаточно интересную программу, моделирующую этот процесс, необходимо привлечь те конструкции языка ассемблера, которые мы еще не обсуждали. Среди прилагаемых к книге файлов в каталоге данной главы находятся исходные тексты подпрограмм, реализующих четыре основных арифметических действия для двоичных операндов произвольной размерности[1]. Не поленитесь внимательно изучить их, так как они являются хорошей иллюстрацией к материалу, изучаемому в этой и последующих главах. К этим примерам можно будет обратиться в полной мере после того, как будут изучены механизмы процедур и макрокоманд (главы 14 и 15). Кроме того, в [8] обсуждаются вопросы программирования целочисленных арифметических операций, но в расширенном контексте.

В завершение обсуждения команд сложения и вычитания отметим, что кроме флагов CF и OF в регистре EFLAGS есть еще несколько флагов, которые можно использовать с двоичными арифметическими командами. Речь идет о следующих флагах:

- ZF — флаг нуля, который устанавливается в 1, если результат операции равен 0, и в 0, если результат не равен 0;
- SF — флаг знака, значение которого после арифметических операций (и не только) совпадает со значением старшего бита результата, то есть с битом 7, 15 или 31 (таким образом, этот флаг можно использовать для операций над числами со знаком).

Умножение двоичных чисел без знака

Для умножения чисел без знака предназначена команда

```
mul сомножитель_1
```

Как видите, в команде указан всего лишь один операнд-сомножитель. Второй операнд-сомножитель задан неявно. Его местоположение фиксировано и зависит от размера сомножителей. Так как в общем случае результат умножения больше, чем любой из его сомножителей, то его размер и местоположение должны быть тоже определены однозначно. Варианты размеров сомножителей и мест размещения второго операнда и результата приведены в табл. 8.2.

Таблица 8.2. Расположение операндов и результата при умножении

Первый сомножитель	Второй сомножитель	Результат
Байт	AL	16 битов в AX: AL — младшая часть результата; AH — старшая часть результата
Слово	AX	32 бита в паре DX:AX: AX — младшая часть результата; DX — старшая часть результата
Двойное слово	EAX	64 бита в паре EDX:EAX: EAX — младшая часть результата; EDX — старшая часть результата

[1] Все прилагаемые к книге файлы можно найти по адресу http://www.piter.com/download. — *Примеч. ред.*

Из таблицы видно, что произведение состоит из двух частей и в зависимости от размера операндов размещается в двух местах — на месте второго сомножителя (младшая часть) и в дополнительных регистрах AH, DX, EDX (старшая часть). Как же динамически (то есть во время выполнения программы) узнать, что результат достаточно мал и уместился в одном регистре или что он превысил размерность регистра и старшая часть оказалась в другом регистре? Для этого привлекаются уже известные нам флаги переноса CF и переполнения OF:

- если старшая часть результата нулевая, то после завершения операции CF = 0 и OF = 0;
- если же флаги CF и OF ненулевые, это означает, что результат вышел за пределы младшей части произведения и состоит из двух частей, что и нужно учитывать при дальнейшей работе.

Рассмотрим следующий пример программы (листинг 8.5).

Листинг 8.5. Умножение

```
<1> ;prg_8_5.asm
<2> masm
<3> model   small
<4> stack   256
<5> .data                       ;сегмент данных
<6> rez label   word
<7> rez_l   db  45
<8> rez_h   db  0
<9> .code                       ;сегмент кода
<10>    main:                   ;точка входа в программу
<11>    ;...
<12>        xor ax,ax
<13>        mov al,25
<14>        mul rez_l
<15>        jnc ml              ;если нет переполнения, то на ml
<16>        mov rez_h,ah ;старшую часть результата в rez_h
<17>    ml:
<18>        mov rez_l,al
<19>        exit:
<20>        mov ax,4c00h;стандартный выход
<21>        int 21h
<22>    end main                ;конец программы
```

В строке 14 производится умножение значения в rez_l на число в регистре AL. Согласно информации из табл. 8.2, результат умножения будет располагаться в регистре AL (младшая часть) и в регистре AH (старшая часть). Для выяснения размера результата в строке 15 командой условного перехода JNC анализируется состояние флага CF, и если оно не равно 1, то результат остается в рамках регистра AL. Если же CF = 1, то выполняется команда в строке 16, которая формирует в поле rez_h старшее слово результата. Команда в строке 18 формирует младшую часть результата. Теперь обратите внимание на сегмент данных, а именно на строку 6. В этой строке содержится директива label. Мы еще не раз будем сталкиваться с этой директивой. В данном случае она назначает еще одно символическое имя rez адресу, на который уже указывает другой идентификатор rez_l. Различие заключается в типах этих идентификаторов — rez имеет тип слова, который ему назначается директивой label (имя типа указано в качестве операнда label). Введя эту директиву в программе, мы подготовились к тому, что, возможно, результат операции

умножения будет занимать в памяти целое слово. Обратите внимание на то, что мы не нарушили принципа: младший байт по младшему адресу. Далее, используя имя rez, можно обращаться к значению в этой области как к слову.

В заключение можно исследовать в отладчике программу на разных наборах сомножителей.

Умножение двоичных чисел со знаком

Для умножения чисел со знаком предназначена команда

```
imul операнд_1[,операнд_2,операнд_3]
```

Эта команда выполняется так же, как и команда MUL. Отличительной особенностью команды IMUL является только формирование знака. Если результат мал и умещается в одном регистре (то есть если CF = OF = 0), то содержимое другого регистра (старшей части) является расширением знака — все его биты равны старшему биту (знаковому разряду) младшей части результата. В противном случае (если CF = OF = 1) знаком результата является знаковый бит старшей части результата, а знаковый бит младшей части является значащим битом двоичного кода результата. Если вы найдете в приложении команду IMUL, то увидите, что у нее имеются более широкие возможности по заданию местоположения операндов. Это сделано для удобства использования.

Деление двоичных чисел без знака

Для деления чисел без знака предназначена команда

```
div делитель
```

Делитель может находиться в памяти или в регистре и иметь размер 8, 16 или 32 бита. Местонахождение делимого фиксировано и так же, как в команде умножения, зависит от размера операндов. Результатом команды деления являются значения частного и остатка. Варианты местоположения и размеров операндов операции деления показаны в табл. 8.3.

Таблица 8.3. Расположение операндов и результата при делении

Делимое	Делитель	Частное	Остаток
Слово (16 бит) в регистре AX	Байт в регистре или в ячейке памяти	Байт в регистре AL	Байт в регистре AH
Двойное слово (32 бита), в DX — старшая часть, в AX — младшая часть	Слово (16 бит) в регистре или в ячейке памяти	Слово (16 бит) в регистре AX	Слово (16 бит) в регистре DX
Учетверенное слово (64 бит), в EDX — старшая часть, в EAX — младшая часть	Двойное слово (32 бита) в регистре или в ячейке памяти	Двойное слово (32 бита) в регистре EAX	Двойное слово (32 бита) в регистре EDX

После выполнения команды деления содержимое флагов неопределенно, но возможно возникновение прерывания с номером 0, называемого «деление на ноль». Этот вид прерывания относится к так называемым *исключениям* и возникает внутри процессора из-за некоторых аномалий в вычислительном процессе. К вопросу

об исключениях мы еще вернемся. Прерывание 0 (деление на ноль) при выполнении команды DIV может возникнуть по одной из следующих причин:

- делитель равен нулю;
- частное не входит в отведенную под него разрядную сетку, что может случиться:
  - при делении делимого величиной в слово на делитель величиной в байт, причем значение делимого более чем в 256 раз больше значения делителя;
  - при делении делимого величиной в двойное слово на делитель величиной в слово, причем значение делимого более чем в 65 536 раз больше значения делителя;
  - при делении делимого величиной в учетверенное слово на делитель величиной в двойное слово, причем значение делимого более чем в 4 294 967 296 раз больше значения делителя.

К примеру, выполним деление значения в области del на значение в области delt (листинг 8.6).

Листинг 8.6. Деление чисел

```
;prg_8.6.asm
masm
model   small
stack   256
.data
del_b   label   byte
del dw  29876
deltdb  45
.code                         ;сегмент кода
    main:                     ;точка входа в программу
    ;...
        xor ax,ax
    ;последующие две команды можно заменить одной mov ax,del
        mov ah,del_b      ;старший байт делимого в ah
        mov al,del_b+1    ;младший байт делимого в al
        div delt          ;в al - частное, в ah - остаток
    ;...
    end main                  ;конец программы
```

Выполнение программы в таком виде приведет к ошибке деления на ноль. Причина описана ранее — частное не входит в отведенную под него разрядную сетку. Это происходит в случаях, когда делимое больше делителя на определенную величину. А что же делать, если соотношение делимое и делителя именно такое? Подробно деление, как, впрочем, и умножение, целых чисел произвольной разрядности описано в книге [8]. Чтобы исправить пример из листинга 8.6, необходимо изменить разрядность делимого, исходя из разрядности делителя и требований команды DIV. К примеру, делимое можно сделать равным 298. Пример будет выполнен без ошибок.

Деление двоичных чисел со знаком

Для деления чисел со знаком предназначена команда

idiv делитель

Для этой команды справедливы все рассмотренные ранее рассуждения, касающиеся команд и чисел со знаком. Отметим лишь особенности возникновения ис-

ключения 0 (деление на ноль) в случае чисел со знаком. Оно возникает при выполнении команды IDIV по одной из следующих причин:

- делитель равен нулю;
- частное не входит в отведенную для него разрядную сетку, что, в свою очередь, может произойти:
  - при делении делимого величиной в слово со знаком на делитель величиной в байт со знаком, причем значение делимого более чем в 128 раз больше значения делителя (таким образом, частное не должно находиться вне диапазона от –128 до +127);
  - при делении делимого величиной в двойное слово со знаком на делитель величиной в слово со знаком, причем значение делимого более чем в 32 768 раз больше значения делителя (таким образом, частное не должно находиться вне диапазона от -32 768 до +32 768);
  - при делении делимого величиной в учетверенное слово со знаком на делитель величиной в двойное слово со знаком, причем значение делимого более чем в 2 147 483 648 раз больше значения делителя (таким образом, частное не должно находиться вне диапазона от -2 147 483 648 до +2 147 483 647).

Вспомогательные команды для арифметических вычислений

В системе команд процессора есть несколько команд, которые облегчают программирование алгоритмов, производящих арифметические вычисления, а также помогают разрешать проблемы, возникающие при подобных вычислениях.

Команды преобразования типов

Что делать, если размеры операндов, участвующих в арифметических операциях, разные? Например, предположим, что в операции сложения один операнд занимает слово, а другой — двойное слово. Ранее было сказано, что в операции сложения должны участвовать операнды одного формата. Если числа без знака, то выход найти просто. В этом случае можно на базе исходного операнда сформировать новый операнд (формата двойного слова), старшие разряды которого просто заполнить нулями. Сложнее ситуация для чисел со знаком: как динамически, в ходе выполнения программы, учесть знак операнда? Для решения подобных проблем в системе команд процессора есть так называемые *команды преобразования типа*. Эти команды расширяют байты в слова, слова — в двойные слова и двойные слова — в учетверенные слова (64-разрядные значения). Команды преобразования типа особенно полезны при преобразовании целых со знаком, так как они автоматически заполняют старшие биты вновь формируемого операнда значениями знакового бита исходного объекта. Эта операция приводит к целым значениям того же знака и той же величины, что и исходная, но уже в более длинном формате. Подобное преобразование называется *операцией распространения знака*.

Существуют два вида команд преобразования типа.

- *Команды без операндов* — эти команды работают с фиксированными регистрами:
  - CBW (Convert Byte to Word) — команда преобразования байта (в регистре AL) в слово (в регистре AX) путем распространения значения старшего бита AL на все биты регистра AH;
  - CWD (Convert Word to Double) — команда преобразования слова (в регистре AX) в двойное слово (в регистрах DX:AX) путем распространения значения старшего бита ax на все биты регистра DX;
  - CWDE (Convert Word to Double) — команда преобразования слова (в регистре AX) в двойное слово (в регистре EAX) путем распространения значения старшего бита AX на все биты старшей половины регистра EAX;
  - CDQ (Convert Double Word to Quarter Word) — команда преобразования двойного слова (в регистре EAX) в учетверенное слово (в регистрах EDX:EAX) путем распространения значения старшего бита EAX на все биты регистра EDX;
- *Команды обработки строк* (также глава 12). Эти команды обладают полезным свойством в контексте нашей проблемы:
  - movsx операнд_1,операнд_2 — команда пересылки с распространением знака расширяет 8- или 16-разрядное значение операнд_2, которое может быть регистром или операндом в памяти, до 16- или 32-разрядного значения в одном из регистров, используя знаковый бит для заполнения старших позиций значения операнд_1 (данную команду удобно использовать для подготовки операндов со знаками к выполнению арифметических операций);
  - movzx операнд_1,операнд_2 — команда пересылки с расширением нулем расширяет 8- или 16-разрядное значение операнд_2 до 16- или 32-разрядного с очисткой (заполнением) нулями старших позиций значения операнд_2 (данную команду удобно использовать для подготовки операндов без знака к выполнению арифметических действий).

К примеру, вычислим значение $y = (a + b)/c$, где a, b, c — байтовые знаковые переменные (листинг 8.7).

Листинг 8.7. Вычисление простого выражения

```
;prg_8_7.asm
masm
model   small
stack   256
.data
a   db  5
b   db  10
c   db  2
y   dw  0
    .code
    main:               ;точка входа в программу
    ;...
        xor ax,ax
        mov al,a
        cbw
  .386
        movsx   bx,b
        add ax,bx
```

```
<18>        idivc           ;в al - частное, в ah - остаток
<19>        exit:
<20>        mov ax,4c00h; стандартный выход
<21>        int 21h
<22>    end main           ;конец программы
```

В этой программе делимое для команды IDIV (строка 18) готовится заранее. Так как делитель имеет размер байта, то делимое должно быть словом. С учетом этого сложение осуществляется параллельно с преобразованием размера результата в слово (строки 14-17). Например, расширение операндов со знаком производится двумя разными командами — CBW и MOVSX.

Другие полезные команды

В системе команд процессора есть две команды — XADD и NEG, которые могут быть полезны, в частности, для программирования вычислительных действий:

- xadd приемник,источник — обмен местами и сложение. Команда позволяет выполнить последовательно два действия:
  1) обменять значения приемник и источник;
  2) поместить на место операнда приемник сумму: приемник = приемник + источник.
- neg операнд — отрицание с дополнением до двух. Команда выполняет инвертирование значения операнд. Физически команда выполняет одно действие: операнд = 0 - операнд, то есть вычитает операнд из нуля. Команду NEG можно применять для смены знака и вычитания из константы. Дело в том, что команды SUB и SBB не позволяют вычесть что-либо из константы, так как константа не может служить операндом-приемником в этих операциях. Поэтому данную операцию можно выполнить с помощью двух команд:

```
neg ax        ;смена знака (ax)
...
add ax,340    ;фактически вычитание: (ax)=340-(ax)
```

Арифметические операции над двоично-десятичными числами

Определение и формат BCD-чисел были рассмотрены в начале этой главы. У вас справедливо может возникнуть вопрос: а зачем нужны BCD-числа? Ответ может быть следующим: BCD-числа нужны в коммерческих приложениях, то есть там, где числа должны быть большими и точными. Как мы уже убедились, выполнение операций с двоичными числами для языка ассемблера довольно проблематично:

- Значения величин в формате слова и двойного слова имеют ограниченный диапазон. Если программа предназначена для работы в области финансов, то ограничение суммы в рублях величиной 65 536 (для слова) или даже 4 294 967 296 (для двойного слова) существенно сужает сферу ее применения (да еще в наших экономических условиях — тут уж никакая деноминация не поможет).
- Двоичные числа дают ошибки округления. Представляете себе программу, работающую где-нибудь в банке, которая не учитывает величину остатка при дей-

ствиях с целыми двоичными числами и оперирует при этом миллиардами. Не хотелось бы быть автором такой программы. Применение чисел с плавающей точкой не спасет — там существует та же проблема округления.

- Большой объем результатов в коммерческих программах требуется представлять в символьном виде (ASCII-коде). Коммерческие программы не просто выполняют вычисления; одной из целей их применения является оперативная выдача информации пользователю. Для этого, естественно, информация должна быть представлена в символьном виде. Перевод чисел из двоичного кода в ASCII-код, как мы уже видели, требует определенных вычислительных затрат. Число с плавающей точкой еще труднее перевести в символьный вид. А вот если посмотреть на шестнадцатеричное представление неупакованной десятичной цифры (в начале данной главы) и на соответствующий ей символ в таблице ASCII, то видно, что они различаются на величину 30h. Таким образом, преобразование в символьный вид и обратно получается намного проще и быстрее.

Эти доводы убеждают в важности овладения хотя бы основами действий с десятичными числами. Далее рассмотрим особенности выполнения основных арифметических операций с такими числами. Для предупреждения возможных вопросов сразу отметим тот факт, что отдельных команд сложения, вычитания, умножения и деления BCD-чисел нет. Сделано это по вполне понятным причинам — размерность таких чисел может быть сколь угодно большой. Складывать и вычитать можно двоично-десятичные числа как в упакованном формате, так и в неупакованном, а вот делить и умножать можно только неупакованные BCD-числа. Почему, будет видно из дальнейшего обсуждения.

Неупакованные BCD-числа

Сложение

Рассмотрим два случая сложения.

Результат сложения не больше 9:

6 = 0000 0110
+
3 = 0000 0011
=
9 = 0000 1001.

Переноса из младшей тетрады в старшую нет. Результат правильный.

Результат сложения больше 9:

06 = 0000 0110
+
07 = 00000111
=
13 = 0000 1101.

То есть мы получили уже не BCD-число. Результат неправильный. Правильный результат в неупакованном BCD-формате в двоичном представлении должен быть таким: 0000 0001 0000 0011 (или 13 в десятичном). Проанализировав данную проблему при сложении BCD-чисел (и подобные проблемы при выполнении дру-

гих арифметических действий), а также возможные пути ее решения, разработчики системы команд процессора решили не вводить специальные команды для работы с BCD-числами, а ввести несколько корректировочных команд. Назначение этих команд — корректировка результата работы обычных арифметических команд для случаев, когда операнды в них являются BCD-числами. В случае сложения во втором примере видно, что полученный результат нужно корректировать. Для коррекции операции сложения двух однозначных неупакованных BCD-чисел и представления результата сложения в символьном виде в системе команд процессора существует специальная команда AAA (ASCII Adjust for Addition).

Команда AAA не имеет операндов. Она работает неявно только с регистром AL и анализирует значение его младшей тетрады. Если это значение меньше 9, то флаг CF сбрасывается в 0 и осуществляется переход к следующей команде. Если это значение больше 9, то выполняются следующие действия.

1. К содержимому младшей тетрады AL (но не к содержимому всего регистра!) прибавляется *6,* тем самым значение десятичного результата корректируется в правильную сторону.
2. Флаг CF устанавливается в 1, тем самым фиксируется перенос в старший разряд для того, чтобы его можно было учесть в последующих действиях.

Так, во втором примере сложения, предполагая, что значение суммы 0000 1101 находится в AL, после выполнения команды AAA в регистре будет 1101 + 0110 = = 0011, то есть двоичное значение 0000 0011 или десятичное значение 3, а флаг CF установится в 1, то есть перенос запоминается в процессоре. Далее программисту нужно использовать команду сложения ADC, которая учтет перенос из предыдущего разряда. Приведем пример программы сложения двух неупакованных BCD-чисел (листинг 8.8).

Листинг 8.8. Сложение неупакованных BCD-чисел

```
<1>     ;prg_8_8.asm
<2>     ;...
<3>     .data
<4>     len equ 2        ;разрядность числа
<5>     b   db  1,7      ;неупакованное число 71
<6>     c   db  4,5      ;неупакованное число 54
<7>     sum db  3 dup (0)
<8>     .code
<9>     main:            ;точка входа в программу
<10>    ;...
<11>        xor bx,bx
<12>        mov cx,len
<13>    m1:
<14>        mov al,b[bx]
<15>        adc al,c[bx]
<16>        aaa
<17>        mov sum[bx],al
<18>        inc bx
<19>        loop m1
<20>        adc sum[bx],0
<21>    ;...
        exit:
```

В листинге 8.8 есть несколько интересных моментов, над которыми стоит поразмыслить. Начнем с описания BCD-чисел. Из строк 5 и 6 видно, что порядок их

ввода обратен нормальному, то есть цифры младших разрядов расположены по меньшему адресу. Это вполне логично по нескольким причинам: во-первых, такой порядок удовлетворяет общему принципу представления данных для процессоров Intel, во-вторых, это очень удобно для поразрядной обработки неупакованных BCD-чисел, так как каждое из них занимает один байт. Хотя, повторюсь, программист сам волен выбирать способ описания BCD-чисел в сегменте данных. Строки 14 и 15 содержат команды, которые складывают цифры в очередных разрядах BCD-чисел, при этом учитывается возможный перенос из младшего разряда. Команда AAA в строке 16 корректирует результат сложения, формируя в AL BCD-цифру и при необходимости устанавливая в 1 флаг CF. Строка 20 учитывает возможность переноса при сложении цифр из самых старших разрядов чисел. Результат сложения формируется в поле sum, описанном в строке 7.

Вычитание

Ситуация при вычитании вполне аналогична сложению. Рассмотрим те же случаи.

Результат вычитания не больше 9:

6 = 0000 0110

–

3 = 00000011

=

3 = 00000011.

Как видим, заема из старшей тетрады нет. Результат верный и корректировки не требует.

Результат вычитания больше 9:

6 =00000110

–

7 =00000111

=

-1-11111111.

Вычитание проводится по правилам двоичной арифметики. Поэтому результат не является BCD-числом. Правильный результат в неупакованном BCD-формате должен быть 9 (0000 1001 в двоичной системе счисления). При этом предполагается заем из старшего разряда, как при обычной команде вычитания, то есть в случае с BCD-числами фактически должно быть выполнено вычитание 16 - 7. Таким образом, как и в случае сложения, результат вычитания нужно корректировать. Для этого существует специальная команда AAS (ASCII Adjust for Substraction), выполняющая коррекцию результата вычитания для представления в символьном виде.

Команда AAS также не имеет операндов и работает с регистром AL, анализируя его младшую тетраду следующим образом: если ее значение меньше 9, то флаг CF сбрасывается в 0 и управление передается следующей команде. Если значение тетрады в AL больше 9, то команда AAS выполняет следующие действия.

1. Из содержимого младшей тетрады регистра AL (заметьте, не из содержимого всего регистра) вычитается 6.

2. Старшая тетрада регистра AL обнуляется.
3. Флаг CF устанавливается в 1, тем самым фиксируется воображаемый заем из старшего разряда.

Понятно, что команда AAS применяется вместе с основными командами вычитания SUB и SBB. При этом команду SUB есть смысл использовать только один раз при вычитании самых младших цифр операндов, далее должна применяться команда SBB, которая будет учитывать возможный заем из старшего разряда. В листинге 8.9 мы обходимся одной командой SBB, которая в цикле производит поразрядное вычитание двух BCD-чисел.

Листинг 8.9. Вычитание неупакованных BCD-чисел

```
<1> ;prg_8_9.asm
<2> masm
<3> model   small
<4> stack   256
<5> .data                           ;сегмент данных
<6> b   db  1,7                     ;неупакованное число 71
<7> c   db  4,5                     ;неупакованное число 54
<8> subs db 2 dup (0)
<9> .code
<10>    main:                       ;точка входа в программу
<11>        mov ax,@data            ;связываем регистр dx с сегментом
<12>        mov ds,ax               ;данных через регистр ax
<13>        xor ax,ax               ;очищаем ax
<14>    len equ 2                   ;разрядность чисел
<15>        xor bx,bx
<16>        mov cx,len              ;загрузка в cx счетчика цикла
<17>    ml:
<18>        mov al,b[bx]
<19>        sbb al,c[bx]
<20>        aas
<21>        mov subs[bx],al
<22>        inc bx
<23>        loop    ml
<24>        jc  m2                  ;анализ флага заема
<25>        jmp exit
<26>    m2: ;...
<27>    exit:
<28>        mov ax,4c00h            ;стандартный выход
<29>        int 21h
<30>    end main                    ;конец программы
```

Данная программа не требует особых пояснений в случае, когда уменьшаемое больше вычитаемого. Поэтому обратите внимание на строку 24. С ее помощью мы предусматриваем случай, когда после вычитания старших цифр чисел был зафиксирован факт заема. Это говорит о том, что вычитаемое было больше уменьшаемого, в результате чего разность оказалась неправильной. Эту ситуацию нужно както обработать. С этой целью в строке 24 командой JC анализируется флаг CF. По результату этого анализа мы уходим на ветку программы, обозначенную меткой т2, где и выполняются нужные действия. Набор этих действий зависит от конкретного алгоритма обработки, поэтому поясним только их суть. Для этого посмотрите в отладчике, как наша программа выполняет вычитание 50 - 74 (правильный ответ——24). То, что вы увидите в окне Dump отладчика (в поле, соответствующем адресу subs), окажется далеким от истины. Что делает в этом случае человек? Он просто мысленно меняет местами вычитаемое и уменьшаемое (74 - 50 = 24), а ре-

зультат рассматривает со знаком «минус». Так и фрагмент программы, обозначенный меткой т2, может, поменяв уменьшаемое и вычитаемое местами и выполнив вычитание, где-то отметить тот факт, что разность, на самом деле, нужно рассматривать как отрицательное число.

Умножение

На примере сложения и вычитания неупакованных чисел мы выяснили, что стандартных алгоритмов для выполнения этих действий над BCD-числами нет и программист должен сам, исходя из требований к своей программе, реализовать эти операции. Реализация двух оставшихся операций — умножения и деления — еще сложнее. В системе команд процессора присутствуют только средства для умножения и деления одноразрядных неупакованных BCD-чисел. Для их умножения необходимо воспроизвести описанную далее процедуру.

1. Поместить один из сомножителей в регистр AL (как того требует команда MUL).
2. Поместить второй сомножитель в регистр или память, отведя для него байт.
3. Перемножить сомножители командой MUL (результат, как и положено, окажется в регистре AX).
4. Скорректировать результат, который, конечно, будет представлен в двоичном коде.

Для коррекции результата после умножения в целях представления его в символьном виде применяется специальная команда AAM (ASCII Adjust for Multiplication). Она не имеет операндов и работает с регистром AX следующим образом.

1. Делит AL на 10.
2. Результат деления записывается так: частное — в AL, остаток — в AH.

В результате после выполнения команды AAM в регистрах AL и AH находятся правильные двоично-десятичные цифры произведения двух цифр.

В листинге 8.10 приведен пример умножения BCD-числа произвольной размерности на однозначное BCD-число.

Листинг 8.10. Умножение неупакованных BCD-чисел

```
masm
model   small
stack   256
.data
b   db  6,7                 ;неупакованное число 76
c   db  4                   ;неупакованное число 4
proizv  db  4 dup (0)
.code
main:                       ;точка входа в программу
        mov ax,@data
        mov ds,ax
        xor ax,ax
    len equ 2               ;размерность сомножителя 1
        xor bx,bx
        xor si,si
        xor di,di
        mov cx,len          ;в cx длина наибольшего сомножителя 1
    ml:
        mov al,b[si]
```

```
        mul c
        aam             ;коррекция умножения
        adc al,dl       ;учли предыдущий перенос
        aaa             ;скорректировали результат сложения с переносом
        mov dl,ah       ;запомнили перенос
        mov proizv[bx],al
        inc si
        inc bx
        loop m1         ;цикл на метку m1
        mov proizv[bx],dl    ;учли последний перенос
        exit:
        mov ax,4c00h
        int 21h
    end main
```

Данную программу можно легко модифицировать для умножения BCD-чисел произвольной длины. Для этого достаточно представить алгоритм умножения в «столбик». Листинг 8.10 можно использовать для получения частичных произведений в этом алгоритме. После их сложения со сдвигом получится искомый результат. Попробуйте написать эту программу самостоятельно. Если же данная задача окажется для вас непосильной, ее решение вы можете найти в [8].

Перед окончанием обсуждения команды AAM необходимо отметить, что ее можно применять для преобразования двоичного числа в регистре AL в неупакованное BCD-число, которое окажется в регистре AX: старшая цифра результата — в AH, младшая — в AL. Понятно, что двоичное число должно быть в диапазоне 0...99.

Деление

Процесс деления двух неупакованных BCD-чисел несколько отличается от других рассмотренных ранее операций с ними. Здесь также требуются действия по коррекции, но они должны выполняться до основной операции, выполняющей непосредственно деление одного BCD-числа на другое BCD-число. Предварительно в регистре AX нужно получить две неупакованные BCD-цифры делимого. Это делает программист удобным для него способом. Далее для коррекции результата деления в целях представления его в символьном виде нужно выполнить команду AAD (ASCII Adjust for Division).

Команда AAD не имеет операндов и преобразует двузначное неупакованное BCD-число в регистре AX в двоичное число, которое впоследствии будет играть роль делимого в операции деления. Кроме преобразования, команда AAD помещает полученное двоичное число в регистр AL. Делимое, естественно, является двоичным числом из диапазона 0...99. Алгоритм, по которому команда AAD осуществляет это преобразование, выглядит следующим образом.

1. Умножить на 10 старшую цифру исходного BCD-числа в AX (содержимое AH).
2. Выполнить сложение AH + AL, результат которого (двоичное число) занести в AL.
3. Обнулить содержимое AH.

Далее программисту нужно выполнить обычную команду деления DIV для деления содержимого AX на одну BCD-цифру, находящуюся в байтовом регистре или байтовой ячейке памяти. Деление неупакованных BCD-чисел иллюстрирует листинг 8.11.

Листинг 8.11. Деление неупакованных BCD-чисел

```
<1> ;prg_8_11.asm
<2> ...
<3> .data            ;сегмент данных
<4> b   db  1,7      ;неупакованное BCD-число 71
<5> c   db  4
<6> .code            ;сегмент кода
<7> main:            ;точка входа в программу
<8> ...
<9>        mov al,b
<10>       aad       ;коррекция перед делением
<11>       div c     ;в al BCD — частное, в ah BCD - остаток
<12>   ...
<13>       exit:
```

Аналогично AAM, команде AAD можно найти и другое применение — использовать ее для перевода неупакованных BCD-чисел из диапазона 0...99 в их двоичный эквивалент.

Для деления чисел большей разрядности так же, как и в случае умножения, нужно реализовывать, например, алгоритм деления в «столбик» или найти более оптимальный путь. Любопытный и настойчивый читатель, возможно, самостоятельно разработает эти программы. Но это делать совсем необязательно. Среди файлов, прилагаемых к книге, в каталоге данной главы приведены тексты макрокоманд, которые выполняют четыре основных арифметических действия с BCD-числами любой разрядности. Кроме этого в [8] вы найдете дополнительный материал по этому вопросу.

Упакованные BCD-числа

Как уже отмечалось ранее, упакованные BCD-числа можно только складывать и вычитать. Для выполнения других действий над ними их нужно дополнительно преобразовывать либо в неупакованный формат, либо в двоичное представление. Таким образом, сами по себе упакованные BCD-числа представляют не слишком большой интерес для программиста, поэтому мы их рассмотрим кратко.

Сложение

Вначале разберемся с сутью проблемы и попытаемся сложить два двузначных упакованных BCD-числа:

67 =01100111
+
75 =01110101
=
142 = 1101 1100 = 220

Как видим, в двоичном виде результат равен 1 101 1 100 (или 220 в десятичном представлении), что неверно. Это происходит по той причине, что процессор не подозревает о существовании BCD-чисел и складывает их по правилам сложения двоичных чисел. На самом деле результат в двоично-десятичном виде должен быть равен 0001 0100 0010 (или 142 в десятичном представлении). Этот пример иллюстрирует необходимость корректировки результатов арифметических операций с упакованными BCD-числами, так же как это было в случае неупакованных BCD-

чисел. Для корректировки результата сложения упакованных чисел в целях представления его в десятичном виде процессор предоставляет команду DAA (Decimal Adjust for Addition).

Команда DAA преобразует содержимое регистра AL в две упакованные десятичные цифры (по алгоритму, приведенному в приложении А, где данная команда описана более подробно). Получившаяся в результате сложения единица (если результат сложения больше 99) запоминается во флаге CF, тем самым учитывается перенос в старший разряд.

Проиллюстрируем сказанное на примере сложения двух двузначных BCD-чисел в упакованном формате (листинг 8.12).

Листинг 8.12. Сложение упакованных BCD-чисел

```
;prg_8_12.asm
...
.data                       ;сегмент данных
b   db  17h                 ;упакованное число 17
c   db  45h                 ;упакованное число 45
sum db  2 dup (0)
.code                       ;сегмент кода
main:                       ;точка входа в программу
...
      mov al,b
      add al,c
      daa
      jnc $+6               ;переход через команду, если результат <= 99
      mov sum+1,ah          ;учет переноса при сложении (результат > 99)
      mov sum,al            ;младшие упакованные цифры результата
   exit:
```

В приведенном примере все достаточно прозрачно; единственное, на что следует обратить внимание, — это описание упакованных BCD-чисел и порядок формирования результата. Результат формируется в соответствии с основным принципом работы процессоров Intel: младший байт по младшему адресу.

Вычитание

Аналогично сложению, при вычитании процессор рассматривает упакованные BCD-числа как двоичные. Выполним вычитание 67 - 75. Так как процессор выполняет вычитание способом сложения, то и мы последуем этому:

67 =01100111
+
-75 = 1011 0101
=
-8 = 0001 1100 = 28.

Как видим, результат равен 28 в десятичной системе счисления, что является абсурдом. В двоично-десятичном коде результат должен быть равен 0000 1000 (или 8 в десятичной системе счисления). При программировании вычитания упакованных BCD-чисел программист, как и при вычитании неупакованных BCD-чисел, должен сам осуществлять контроль за знаком. Это делается с помощью флага CF, который фиксирует заем из старших разрядов. Само вычитание BCD-чисел осуществляется обычной командой вычитания SUB или SBB. Коррекция результата

вычитания для его представления в десятичном виде осуществляется командой DAS (Decimal Adjust for Substraction).

В приложении описан алгоритм, по которому команда DAS преобразует содержимое регистра AL в две упакованные десятичные цифры.

Итоги

- Процессор имеет довольно мощные средства для реализации вычислительных операций. Для этого у него есть блок целочисленных операций и блок операций с плавающей точкой. Для большинства задач, использующих язык ассемблера, достаточно целочисленной арифметики.
- Команды целочисленных операций работают с данными двух типов: двоичными и двоично-десятичными числами (BCD-числами).
- Двоичные данные могут либо иметь знак, либо не иметь такового. Процессор, на самом деле, не различает числа со знаком и без. Он лишь помогаетотслеживать изменение состояния некоторых битов операндов и состояние отдельных флагов. Операции сложения и вычитания чисел со знаком и без знака проводятся одним устройством и по единым правилам.
- Контроль за правильностью результатов и их надлежащей интерпретацией полностью лежит на программисте. Он должен контролировать состояние флагов CF и OF регистра EFLAGS во время вычислительного процесса.
- Для операций с числами без знака нужно контролировать флаг CF. Установка его в 1 сигнализирует о том, что число вышло за разрядную сетку операндов.
- Для чисел со знаком установка флага OF в 1 говорит о том, что в результате сложения чисел одного знака результат выходит за границу допустимых значений чисел со знаком в данном формате, и сам результат меняет знак (пропадает порядок).
- По результатам выполнения арифметических операций устанавливаются также флаги PF, ZF и SF.
- В отличие от команд сложения и вычитания, команды умножения и деления позволяют учитывать знаки операндов.
- Арифметические команды очень «капризны» к размерности операндов, поэтому в систему команд процессора включены специальные команды, позволяющие отслеживать эту характеристику.
- Хотя диапазон значений двоичных данных довольно велик, для коммерческих приложений его явно недостаточно, поэтому в архитектуру процессора введены средства для работы с так называемыми двоично-десятичными (BCD) данными.
- Двоично-десятичные данные представляются в двух форматах, упакованном и неупакованном. Наиболее универсальным является неупакованный формат.

Глава 9

Логические команды и команды сдвига

- **Краткое описание группы логических команд**
- **Команды для выполнения логических операций**
- **Команды сдвига**
- **Организация работы с отдельными битами**

Наряду со средствами арифметических вычислений система команд процессора имеет также средства логического преобразования данных. Под *логическим* понимается такое преобразование данных, в основе которого лежат правила *формальной логики*. Формальная логика работает на уровне утверждений *истинно* или *ложно*. Для процессора это, как правило, означает 1 или 0 соответственно. Для компьютера язык нулей и единиц является родным, но минимальной единицей данных, с которой работают машинные команды, является байт. Однако на системном уровне часто необходимо иметь возможность работать на предельно низком уровне — на уровне битов.

К средствам логического преобразования данных в языке ассемблера относятся *логические команды* и *логические операции* (рис. 9.1). Команды рассматриваются в этой главе, операции были изучены нами в главе 5. Напомню, что операнд команды ассемблера в общем случае может представлять собой выражение, которое, в свою очередь, является комбинацией операторов и операндов. Среди этих операторов могут быть и операторы, реализующие логические операции над объектами выражения.

Перед знакомством с этими средствами давайте посмотрим, что же представляют собой сами логические данные и какие действия над ними могут производиться.

7 Зак. 256

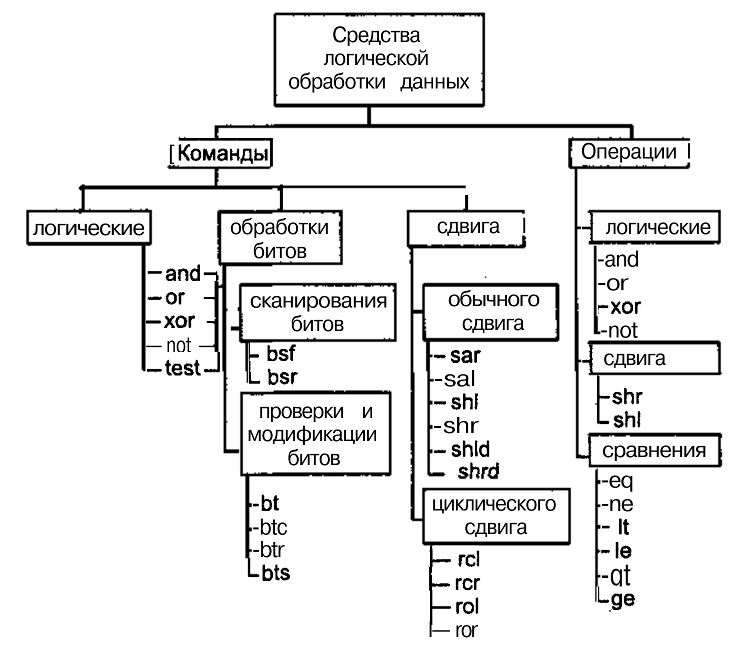


Рис. 9.1. Средства процессора для работы с логическими данными

Логические данные

Теоретической базой для логической обработки данных является формальная логика. Существует несколько систем логики. Одна из наиболее известных — это *исчисление высказываний. Высказывание* — это любое утверждение, о котором можно сказать, что оно либо *истинно,* либо *ложно.* Исчисление высказываний представляет собой совокупность правил определения истинности или ложности некоторой комбинации высказываний.

Исчисление высказываний очень гармонично сочетается с принципами работы компьютера и основными методами его программирования. Все аппаратные компоненты компьютера построены на логических микросхемах. Система представления информации в компьютере на самом нижнем уровне основана на понятии *бита.* Бит, имея всего два состояния — 0 (ложно) и 1 (истинно), естественным образом вписывается в систему исчисления высказываний.

Согласно теории, над высказываниями (над битами) могут выполняться *логические операции.*

- *Отрицание* (логическое *НЕ)* — логическая операция над одним операндом, результатом которой является величина, обратная значению исходного операнда. Эта операция однозначно характеризуется следующей *таблицей истинности*[1]:

[1] Таблица истинности — таблица результатов логических операций в зависимости от значений исходных операндов.

Значение операнда:	0	1
Результат операции:	1	0

- *Логическое сложение* (логическое *включающее ИЛИ)* — логическая операция над двумя операндами, результатом которой является истина (1), если один или оба операнда истинны (1), или ложь (0), если оба операнда ложны (0). Эта операция описывается с помощью следующей таблицы истинности:

Значение операнда 1:	0	0	1	1
Значение операнда 2:	0	1	0	1
Результат операции:	0	1	1	1

- *Логическое умножение* (логическое *И)* — логическая операция над двумя операндами, результатом которой является истина (1) только в том случае, если оба операнда истинны (1). Во всех остальных случаях значение операции — ложь (0). Эта операция описывается с помощью следующей таблицы истинности:

Значение операнда 1 :	0	0	1	1
Значение операнда 2:	0	1	0	1
Результат операции:	0	0	0	1

- *Логическое исключающее сложение* (логическое *исключающее ИЛИ)* — логическая операция над двумя операндами, результатом которой является истина (1), если только один из двух операндов истинен (1), и ложь (0), если оба операнда либо ложны (0), либо истинны (1). Эта операция описывается с помощью следующей таблицы истинности:

Значение операнда 1:	0	0	1	1
Значение операнда 2:	0	1	0	1
Результат операции:	0	1	1	0

Логические команды

Система команд процессора содержит пять команд, поддерживающих описанные ранее операции. Эти команды выполняют логические операции над битами операндов. Размерность операндов, естественно, должна быть одинакова. Например, если размерность операндов равна слову (16 битов), то логическая операция выполняется сначала над нулевыми битами операндов, и ее результат записывается на место бита 0 результата. Далее команда последовательно повторяет эти действия над всеми битами с первого до пятнадцатого. Возможные варианты размерности операндов для каждой команды можно найти в приложении.

Далее перечислены базовые команды процессора, поддерживающие работу с логическими данными:

- and `операнд_1,операнд_2` — операция логического умножения. Команда выполняет поразрядно логическую операцию *И* (*конъюнкцию*) над битами операндов `операнд_1` и `операнд_2`. Результат записывается на место `операнд_1`.
- or `операнд_1,операнд_2` — операция логического сложения. Команда выполняет поразрядно логическую операцию *ИЛИ (дизъюнкцию)* над битами операндов `операнд_1` и `операнд_2`. Результат записывается на место `операнд_1`.

- xor операнд_1,операнд_2 — операция логического исключающего сложения. Команда выполняет поразрядно логическую операцию исключающего *ИЛИ* над битами операндов операнд_1 и операнд_2. Результат записывается на место операнд_1.
- test операнд_1,операнд_2 — операция проверки (способом логического умножения). Команда выполняет поразрядно логическую операцию *И* над битами операндов операнд_1 и операнд_2. Состояние операндов остается прежним, изменяются только флаги ZF, SF, и PF, что дает возможность анализировать состояние отдельных битов операнда без изменения их состояния в исходных операндах.
- not операнд — операция логического отрицания. Команда выполняет поразрядное инвертирование (замену значения на обратное) каждого бита операнда. Результат записывается на место операнда.

Для представления роли логических команд в системе команд процессора очень важно понять области их применения и типовые приемы их использования при программировании. Далее мы будем рассматривать логические команды в контексте обработки последовательности битов.

Очень часто некоторая ячейка памяти должна играть роль индикатора, показывая, например, занятость некоторого программного или аппаратного ресурса. Так как эта ячейка может принимать только два значения — занято (1) или свободно (0), то отводить под нее целый байт очень расточительно, логичнее для этой цели использовать бит. А если таких индикаторов много? Объединив их в пределах одного байта или слова, можно получить довольно существенную экономию памяти. Посмотрим, что могут сделать для этого логические команды.

С помощью логических команд возможно выделение отдельных битов в операнде с целью их установки, сброса, инвертирования или просто проверки на определенное значение. Для организации подобной работы с битами второй операнд обычно играет роль *маски*. Путем установки в 1 битов этой маски и определяются нужные для конкретной операции биты первого операнда. Покажем, какие логические команды могут применяться для этой цели.

Для установки определенных разрядов (битов) в 1 применяется команда

```
or операнд_1,операнд_2
```

В этой команде второй операнд, играющий роль маски, должен содержать единичные биты на месте тех разрядов, которые должны быть установлены в 1 в первом операнде:

```
or  eax,10b ;установить 1-й бит в регистре eax
```

Для сброса определенных разрядов (битов) в 0 применяется команда

```
and операнд_1,операнд_2
```

В этой команде второй операнд, играющий роль маски, должен содержать нулевые биты на месте тех разрядов, которые должны быть установлены в 0 в первом операнде:

```
and  eax,0fffffffdh ;сбросить в 0 1-й бит в регистре eax
```

Для выяснения того, какие биты в обоих операндах различаются, или для инвертирования заданных битов в первом операнде применяется команда

```
xor операнд_1,операнд_2
```

Интересующие нас биты маски (второго операнда) при выполнении команды XOR должны быть единичными, остальные — нулевыми:

```
xor eax,10b ;инвертировать 1-й бит в регистре eax
jz  mes     ;переход, если 1-й бит в al был единичным
```

Для проверки состояния заданных битов в первом операнде применяется команда

```
test операнд_1,операнд_2
```

Проверяемые биты первого операнда в маске (втором операнде) должны иметь единичное значение. Алгоритм работы команды TEST подобен алгоритму работы команды AND, но он не меняет значения первого операнда. Результатом команды является установка значения флага нуля ZF:

- если ZF = 0, то в результате логического умножения получился ненулевой результат, то есть хотя бы один единичный бит маски совпал с соответствующим единичным битом первого операнда;
- если ZF = 1, то в результате логического умножения получился нулевой результат, то есть ни один единичный бит маски не совпал с соответствующим единичным битом первого операнда.

Таким образом, если любые соответствующие биты в обоих операндах установлены, то ZF = 0. Для реакции на результат команды TEST целесообразно использовать команду перехода на метку JNZ (Jump if Not Zero) — переход, если флаг нуля ZF ненулевой, или команду с обратным действием JZ (Jump if Zero) — переход, если флаг нуля ZF нулевой. Например,

```
testeax,00000010h
jnz ml   ;переход если 4-й бит равен 1
```

Начиная с системы команд процессора i386, набор команд для поразрядной обработки данных расширился. При использовании этих команд необходимо указывать одну из директив: .386, .486 и т. д. Следующие две команды позволяют осуществить поиск первого установленного в 1 бита операнда. Поиск можно произвести как с начала, так и от конца операнда:

- bsf операнд_1,операнд_2 — сканирование битов вперед (Bit Scaning Forward). Команда просматривает (сканирует) биты второго операнда от младшего к старшему (от бита 0 до старшего бита) в поисках первого бита, установленного в 1. Если таковой обнаруживается, в первый операнд заносится номер этого бита в виде целочисленного значения. Если все биты второго операнда равны 0, то флаг нуля ZF устанавливается в 1, в противном случае флаг ZF сбрасывается в 0.

```
mov al,02h
bsf bx,al    ;bx=l
jz  ml  ;переход, если al=00h
...
```

- bsr операнд_1,операнд_2 — сканирование битов в обратном порядке (Bit Scaning Reset). Команда просматривает (сканирует) биты второго операнда от старшего к младшему (от старшего бита к биту 0) в поисках первого бита, установленного в 1. Если таковой обнаруживается, в первый операнд заносится номер этого бита в виде целочисленного значения. При этом важно, что позиция первого единичного бита слева все равно отсчитывается относительно бита 0. Если все биты второго операнда равны 0, то флаг нуля ZF устанавливается в 1, в противном случае флаг ZF сбрасывается в 0.

Листинг 9.1 демонстрирует пример применения команд BSR и BSF. Введите код и исследуйте работу программы в отладчике (в частности, обратите внимание на то, как меняется содержимое регистра BX после выполнения команд BSF и BSR).

Листинг 9.1. Сканирование битов

```
;prg_9_1.asm
masm
model   small
stack   256
.data                   ;сегмент данных
.code                   ;сегмент кода
main:                   ;точка входа в программу
    mov ax,@data
    mov ds,ax
;...
.486                    ;это обязательно
    xor ax,ax
    mov al,02h
    bsf bx,ax           ;bx=l
    jz  ml              ;переход, если al=00h
    bsr bx,ax
ml:
;...
    mov ax,4c00h;стандартный выход
    int 21h
end main
```

Интерес представляют еще несколько из группы логических команд, позволяющих реализовать доступ к конкретному биту операнда. Они, как и предыдущие, появились в моделях процессоров Intel, начиная с i386. Поэтому при их использовании не забывайте указывать одну из директив: .386, .486 и т. д. Операнд может находиться как в памяти, так и в регистре общего назначения. Положение бита задается смещением его относительно младшего бита операнда. Смещение может как задаваться в виде непосредственного значения, так и содержаться в регистре общего назначения. В качестве значения смещения вы можете использовать результаты работы команд BSR и BSF. Все команды присваивают значение выбранного бита флагу CF.

Команда проверки бита BT (Bit Test) переносит значение бита в флаг CF:

```
bt операнд,смещение_бита
```

Например,

```
bt  ax,5    ;проверить значение бита 5
jnc ml      ;переход, если бит = 0
```

Команда проверки и установки бита BTS (Bit Test and Set) переносит значение бита в флаг CF и затем устанавливает проверяемый бит в 1:

```
bts операнд,смещение_бита
```

Например,

```
mov ax,10
bts pole,ax ;проверить и установить 10-й бит в pole
jc  ml      ;переход, если проверяемый бит был равен 1
```

Команда проверки и сброса бита BTR (Bit Test and Reset) переносит значение бита во флаг CF и затем устанавливает этот бит в 0:

```
btr операнд,смещение_бита
```

Команда проверки и инвертирования бита BTC (Bit Test and Convert) переносит значение бита в флаг CF и затем инвертирует значение этого бита:

```
btc операнд,смещение_бита
```

Команды сдвига

Команды сдвига также обеспечивают манипуляции над отдельными битами операндов, но иным способом, чем рассмотренные ранее логические команды. Все команды сдвига перемещают биты в поле операнда влево или вправо, в зависимости от кода операции. Все команды сдвига имеют одинаковую структуру:

```
коп операнд,счетчик_сдвигов
```

Количество сдвигаемых разрядов (значение счетчик_сдвигов) может задаваться двумя способами:

- статически — непосредственно во втором операнде;
- динамически — в регистре CL перед выполнением команды сдвига.

Исходя из размерности регистра CL, понятно, что значение счетчик_сдвигов может лежать в диапазоне от 0 до 255. Но на самом деле это не совсем так. В целях оптимизации процессор воспринимает только значения пяти младших битов счетчика, то есть значение лежит в диапазоне от 0 до 31. В последних моделях процессора есть дополнительные команды, позволяющие делать 64-разрядные сдвиги. Мы их рассмотрим чуть позже.

Все команды сдвига устанавливают флаг переноса CF. По мере сдвига битов за пределы операнда они сначала попадают во флаг переноса CF, устанавливая его равным значению очередного бита, оказавшегося за пределами операнда. Куда этот бит попадет дальше, зависит от типа команды сдвига и алгоритма программы.

По принципу действия команды сдвига можно разделить на два типа:

- команды *линейного сдвига;*
- команды *циклического сдвига.*

Линейный сдвиг

К командам линейного сдвига относятся команды, осуществляющие сдвиг по следующему алгоритму.

1. Очередной «выдвигаемый» бит устанавливает флаг CF.
2. Бит, появляющийся с другого конца операнда, имеет значение 0.
3. При сдвиге очередного бита он переходит во флаг CF, при этом значение предыдущего сдвинутого бита *теряется*!

Команды линейного сдвига делятся на два подтипа:

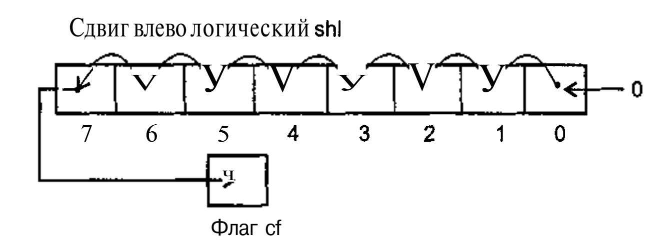
- команды *логического* линейного сдвига;
- команды *арифметического* линейного сдвига.

Далее перечислены команды логического линейного сдвига:

- `shl операнд,счетчик_сдвигов` — логический сдвиг влево (Shift Logical Left). Содержимое операнда сдвигается влево на количество битов, определяемое значением `счетчик_сдвигов`. Справа в позицию младшего бита вписываются нули.

- shr операнд,счетчик_сдвигов — логический сдвиг вправо (Shift Logical Right). Содержимое операнда сдвигается вправо на количество битов, определяемое значением счетчик_сдвигов. Слева в позицию старшего (знакового) бита вписываются нули.

Рисунок 9.2 иллюстрирует принцип работы этих команд.



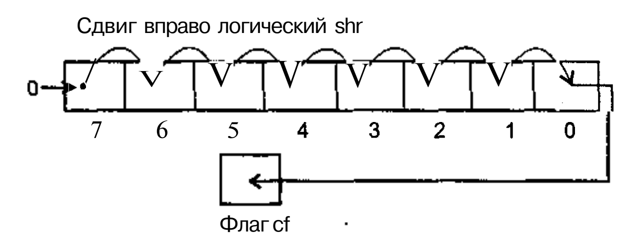


Рис. 9.2. Схема работы команд линейного логического сдвига

Ниже показан фрагмент программы, выполняющей преобразование двух неупакованных BCD-чисел в слове памяти bcd_dig в упакованное BCD-число в регистре AL.

```
...
bcd_dig dw  0905h    ;описание неупакованного BCD-числа 95
...
    mov ax,bcd_dig   ;пересылка
    shl ah,4         ;сдвиг влево
    add al,ah        ;сложение для получения результата: al=95h
```

Команды арифметического линейного сдвига отличаются от команд логического сдвига тем, что они особым образом работают со знаковым разрядом операнда:

- sal операнд,счетчик_сдвигов — арифметический сдвиг влево (Shift Arithmetic Left). Содержимое операнда сдвигается влево на количество битов, определяемое значением счетчик_сдвигов. Справа (в позицию младшего бита) вписываются нули. Команда SAL не сохраняет знака, но устанавливает флаг OF в случае смены знака очередным выдвигаемым битом. В остальном команда SAL полностью аналогична команде SHL;
- sar операнд,счетчик_сдвигов — арифметический сдвиг вправо (Shift Arithmetic Right). Содержимое операнда сдвигается вправо на количество битов, определяемое значением счетчик_сдвигов. Слева в операнд вписываются нули. Команда SAR сохраняет знак, восстанавливая его после сдвига каждого очередного бита.

Рисунок 9.3 иллюстрирует принцип работы команд линейного арифметического сдвига.

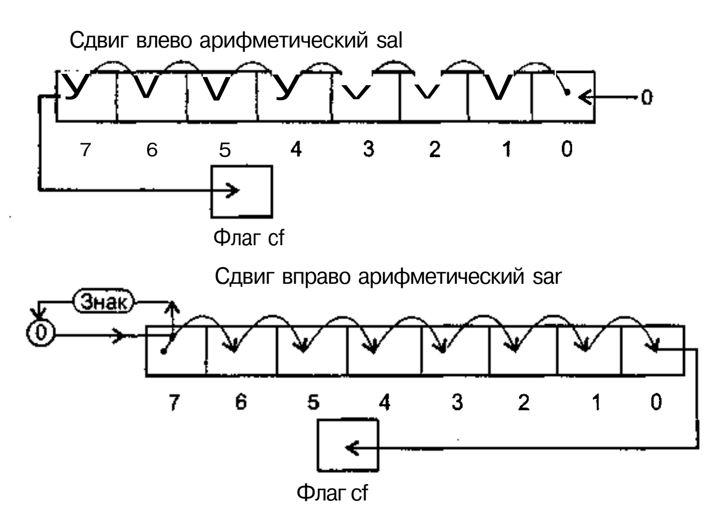


Рис. 9.3. Схема работы команд линейного арифметического сдвига

Команды арифметического сдвига позволяют выполнить «быстрое» умножение и деление операнда на степени двойки. Посмотрите на двоичное представление чисел 75 и 150:

75 01001011
150 10010110

Второе число является сдвинутым влево на один разряд первым числом. Если у вас еще есть сомнения, проделайте несколько умножений на 2, 4, 8 и т. д.

Аналогичная ситуация — с операцией деления. Сдвигая вправо операнд, мы, фактически, осуществляем операцию деления на степени двойки 2, 4, 8 и т. д.

Преимущество этих команд по сравнению с командами умножения и деления — в скорости их исполнения процессором, что может пригодиться при оптимизации программы.

Циклический сдвиг

К командам циклического сдвига относятся команды, сохраняющие значения сдвигаемых битов. Есть два типа команд циклического сдвига:

- команды простого циклического сдвига (рис. 9.4);
- команды циклического сдвига через флаг переноса CF (рис. 9.5).

Далее перечислены команды простого циклического сдвига:

- rol операнд,счетчик_сдвигов — циклический сдвиг влево (Rotate Left). Содержимое операнда сдвигается влево на количество битов, определяемое операндом счетчик_сдвигов. Сдвигаемые влево биты записываются в тот же операнд справа.
- ror операнд,счетчик_сдвигов — циклический сдвиг вправо (Rotate Right). Содержимое операнда сдвигается вправо на количество битов, определяемое операндом счетчик_сдвигов. Сдвигаемые вправо биты записываются в тот же операнд слева.

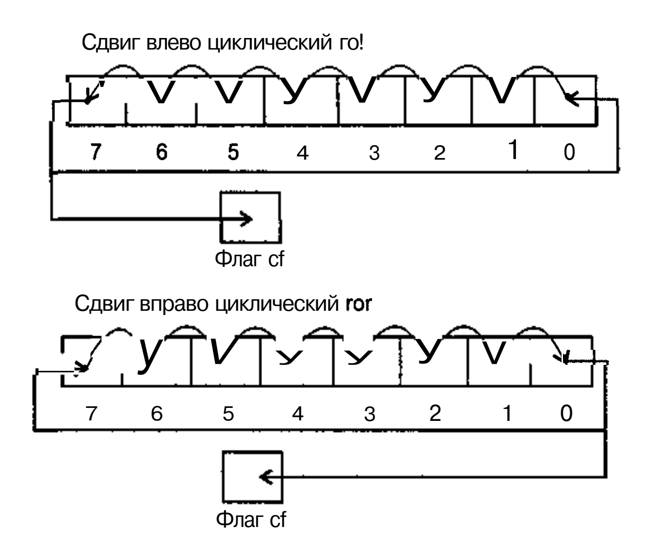


Рис. 9.4. Схема работы команд простого циклического сдвига

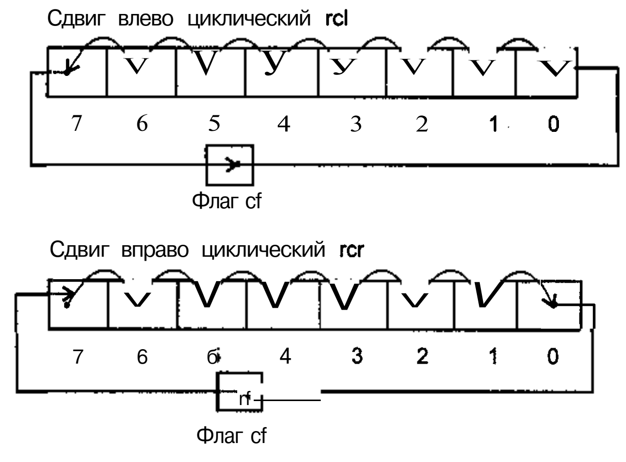


Рис. 9.5. Схема работы команд циклического сдвига через флаг переноса CF

Как видно из рис. 9.4, команды простого циклического сдвига в процессе своей работы осуществляют одно полезное действие: циклически сдвигаемый бит не только вдвигается в операнд с другого конца, но и одновременно его значение становится значением флага CF. К примеру, для того чтобы обменять содержимое двух половинок регистра EAX, достаточно выполнить следующую последовательность команд:

```
mov eax,ffff0000h
mov cl,16
rol eax,cl
```

Команды циклического сдвига через флаг переноса CF отличаются от команд простого циклического сдвига тем, что сдвигаемый бит не сразу попадает в операнд с другого его конца, а сначала записывается во флаг переноса CF. Лишь следу-

ющее исполнение данной команды сдвига (при условии, что она выполняется в цикле) приводит к помещению выдвинутого ранее бита в другой конец операнда (рис. 9.5). Команды циклического сдвига через флаг переноса CF перечислены далее:

- rcl операнд,счетчик_сдвигов — циклический сдвиг влево через перенос (Rotate through Carry Left). Содержимое операнда сдвигается влево на количество битов, определяемое операндом счетчик_сдвигов. Сдвигаемые биты поочередно становятся значением флага переноса CF;
- rcr операнд,счетчик_сдвигов — циклический сдвиг вправо через перенос (Rotate through Carry Right). Содержимое операнда сдвигается вправо на количество битов, определяемое операндом счетчик_сдвигов. Сдвигаемые биты поочередно становятся значением флага переноса CF.

Из рис. 9.5 видно, что при сдвиге через флаг переноса появляется промежуточный элемент, с помощью которого можно производить подмену циклически сдвигаемых битов, в частности, выполнять рассогласование битовых последовательностей. Под рассогласованием битовой последовательности здесь и далее подразумевается действие, которое позволяет некоторым образом локализовать и извлечь нужные участки этой последовательности и записать их в другое место. Например, рассмотрим, как переписать в регистр BX старшую половину регистра EAX с одновременным ее обнулением в регистре EAX:

```
    mov cx,16       ;кол-во сдвигов для eax
ml:
    clc             ;сброс флага cf в 0
    rcl  eax,1      ;сдвиг крайнего левого бита из eax в cf
    rcl bx,1        ;перемещение бита из cf справа в bx
    loopml          ;цикл 16 раз
    rol eax,16      ;восстановить правую часть eax
```

Команды простого циклического сдвига можно использовать для операций другого рода. К примеру, подсчитаем количество единичных битов в регистре EAX:

```
    xor dx,dx       ;очистка dx для подсчета единичных битов
    mov cx,32       ;число циклов подсчета
cycl:               ;метка цикла
    ror eax,1       ;циклический сдвиг вправо на 1 бит
    jnc  not_one    ;переход, если очередной бит в cf
                    ;не равен единице
    inc  dx         ;увеличение счетчика цикла
not_one:
    loopcycl        ;переход на метку cycl, если
                    ;значение в cx не равно 0
```

Этот фрагмент не требует особых пояснений, единственное, что нужно помнить, — особенности работы команды цикла ШОР (глава 10). Команда ШОР сравнивает значение регистра ECX/CX с нулем и, если оно не равно нулю, выполняет уменьшение ECX/CX на единицу и передает управление на метку в программе, указанную в этой команде в качестве операнда.

Дополнительные команды сдвига

Система команд моделей микропроцессоров Intel, начиная с 80386, содержит дополнительные команды сдвига, расширяющие рассмотренные нами ранее возможности. Это — *команды сдвига двойной точности:*

- shld операнд_1,операнд_2,счетчик_сдвигов — сдвиг влево двойной точности. Команда сдвигает биты первого операнда влево и заполняет его справа значениями битов, вытесняемых из второго операнда, согласно схеме на рис. 9.6. Количество сдвигаемых битов определяется значением счетчика сдвигов, которое может лежать в диапазоне 0...31. Это значение может задаваться непосредственно в третьем операнде или содержаться в регистре CL. Значение второго операнда не меняется;

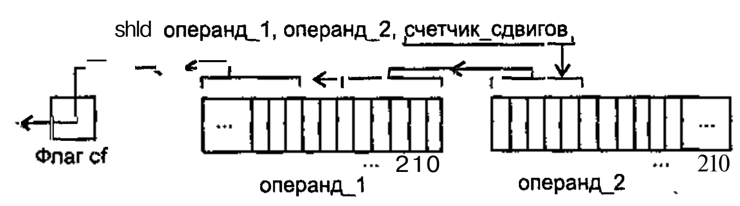


Рис. 9.6. Схема работы команды SHLD

- shrd операнд_1,операнд_2,счетчик_сдвигов — сдвиг вправо двойной точности. Команда сдвигает биты первого операнда вправо и заполняет его слева значениями битов, вытесняемых из второго операнда, согласно схеме на рис. 9.7. Количество сдвигаемых битов определяется значением счетчика сдвигов, которое может лежать в диапазоне 0...31. Это значение может задаваться непосредственно в третьем операнде или содержаться в регистре CL. Значение второго операнда не меняется.

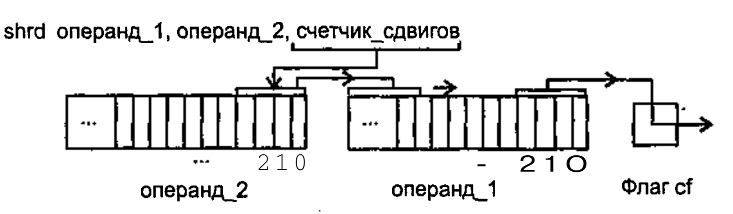


Рис. 9.7. Схема работы команды SHRD

Как мы отметили, команды SHLD и SHRD осуществляют сдвиги на величину до 32 разрядов, но за счет особенностей задания операндов и алгоритма работы эти команды можно использовать для работы с полями длиной до 64 битов. Например, рассмотрим, как можно осуществить сдвиг влево на 16 битов поля из 64 битов.

```
;...
.data
pole_l  dd  0b21187f5h
pole_h  dd  45ff6711h
.code
;...
.386
    mov cl,16   ;загрузка счетчика сдвигов в cl
    mov eax,pole_h
    shldpole_l,eax,cl
    shl pole_h,cl   ;pole_l=87f50000h;
pole_h=6711b211h
```

Рассмотрим еще некоторые наиболее типичные примеры применения этих команд. Отметим следующий момент. Рассмотренные далее действия, конечно, можно

выполнить и множеством других способов, но эти являются самыми быстрыми. Если ваши программы должны работать максимально быстро, то есть смысл потратить время на разбор этих примеров.

Примеры работы с битовыми строками

Рассогласование битовых строк

Наглядный пример рассогласования последовательностей битов — преобразование неупакованного BCD-числа в упакованное BCD-число. Один из вариантов такого преобразования был рассмотрен нами ранее при обсуждении команды линейного сдвига SHL. Попробуем выполнить подобное преобразование с использованием команд сдвига двойной точности (листинг 9.2). В общем случае длина числа может быть произвольной, но при этом нужно учитывать ограничения, которые накладываются используемыми ресурсами процессора. Ограничения связаны в основном с тем, что центральное место в преобразовании занимает регистр EAX, поэтому, если преобразуемое число имеет размер более четырех байтов, то его придется делить на части. Но это уже чисто алгоритмическая задача, поэтому в нашем случае предполагается, что неупакованное BCD-число имеет длину 4 байта.

Листинг 9.2. Преобразование BCD-числа (вариант 2)

```
;prg_9_2.asm
masm
model   small
stack   256
.data
len=4                           ;длина неупакованного BCD-числа
unpck_BCD     label    dword
dig_BCD  db  2,4,3,6           ;неупакованное BCD-число 6342
pck_BCD  dd  0                  ;pck_BCD=00006342
.code
main:                           ;точка входа в программу
    mov  ax,@data
    mov  ds,ax
    xor  ax,ax
    mov  cx,len
.386                            ;это обязательно
    mov  eax,unpck_BCD
ml:
    shl  eax,4                  ;убираем нулевую тетраду
    shld pck_BCD,eax,4          ;тетраду с цифрой заносим в поле pck_BCD
    shl  eax,4                  ;убираем тетраду с цифрой из eax
    loop ml                     ;цикл
    exit:                        ;pck_BCD=00006342
    mov  ax,4c00h
    int  21h
end     main
```

Команды сдвига двойной точности SHLD и SHRD позволяют осуществлять с максимально возможной скоростью вставку битовой строки из регистра в произвольное место другой (большей) строки битов в памяти и извлечение в регистр битовой подстроки из некоторой строки битов в памяти. В результате этих операций смежные с подстрокой биты по ее обеим сторонам остаются неизменными.

Вставка битовых строк

Рассмотрим пример вставки битовой строки длиной 16 битов, находящейся в регистре EAX, в строку памяти str, начиная с ее бита 8 (листинг 9.3). Вставляемая битовая строка выровнена к левому краю регистра EAX.

Листинг 9.3. Вставка битовой строки

```
<1>     ;prg_9_3.asm
<2>     raasm
<3>     model   small
<4>     stack   256
<5>     .data
<6>     bit_str dd  11010111h   ;строка для вставки
<7>     P_str   dd  0ffff0000h  ;вставляемая подстрока 0ffffh
<8>     .code
<9>     main:                   ;точка входа в программу
<10>        mov ax,@data
<11>        mov ds,ax
<12>        xor ax,ax
<13>    .386                    ;это обязательно
<14>        mov eax,p_str
<15>    ;правый край места вставки циклически переместить к краю
<16>    ;строки bit_str (сохранение правого контекста):
<17>        ror bit_str,8
<18>        shr bit_str,16      ;сдвинуть строку вправо
                                ;на длину подстроки (16 битов)
<19>        shld bit_str,eax,16 ;сдвинуть 16 бит
<20>        rol bit_str,8       ;восстановить младшие 8 бит
<21>    ;...
<22>        exit:               ;bit_str=11ffff11
<23>        mov ax,4c00h
<24>        int 21h
<25>    end main
```

Листинг 9.3 удобно исследовать в отладчике. При этом важно понять зависимость между непосредственными значениями, используемыми в командах строк 17-20, и исходными значениями. Общая методика вставки битовых строк выглядит следующим образом.

1. Подогнать к правому краю строки младший бит места вставки в этой строке. Делать это нужно командой циклического сдвига, чтобы сохранить правую часть исходной строки. Величина сдвига определяется очень просто — это номер начальной позиции места вставки (строка 17).
2. Сдвинуть исходную строку вправо на количество битов, равное длине вставляемой подстроки (строка 18). Эти биты нам больше не нужны, поэтому для сдвига используется команда простого сдвига SHR.
3. Командой SHLD вставить вставляемую подстроку в исходную подстроку. Перед этим, естественно, левый край вставляемой подстроки находится у левого края регистра EAX (строка 19).
4. Восстановить командой циклического сдвига правую часть исходной строки (строка 20).

Наибольшей эффективности при использовании этой программы можно достичь, если оформить представленную в ней последовательность команд в виде макрокоманды. Понятие макрокоманды будет рассматриваться нами в главе 14, но сейчас важно отметить, что в данном случае она позволит нам не задумываться

о настройке строк 17–20 на конкретную вставку. При изучении материала главы 14 вы можете поэкспериментировать с данной программой, разработав на ее основе макрокоманду.

Извлечение битовых строк

Рассмотрим пример извлечения 16 битов из строки в памяти bit_str, начиная с бита 8, в регистр EAX (листинг 9.4). Результат следует выровнять по правому краю регистра EAX; строка bit_str не изменяется. Этот пример можно рассматривать как обратный тому, который мы только что привели в листинге 9.3. Методика извлечения битовой подстроки, если вы разобрались с программой вставки битовой строки, не должна вызвать у вас затруднений.

Листинг 9.4. Извлечение битовой строки

```
;prg_9_4.asm
masm
model   small
stack  256
.data
bit_str dd  11ffff11h   ;строка для извлечения
.code
main:                   ;точка входа в программу
    mov ax,@data
    mov ds,ax
    xor ax,ax
.386                    ;это обязательно
;левый край места извлечения циклически переместить к левому краю
;строки bit_str (сохранение левого контекста)
    rol bit_str,8
    mov ebx,bit_str     ;подготовленную строку в ebx
    shld eax,ebx,16     ;вставить извлекаемые 16 бит
    ;в регистр eax
    ror bit_str,8       ;восстановить старшие 8 бит
;...
    exit:               ;eax=0000ffff
    mov ax,4c00h
    int 21h
end  main
```

Пересылка битов

По сути, программа пересылки битов является комбинацией двух предыдущих. Поэтому попробуйте самостоятельно разработать программу пересылки блока битов из одной битовой строки в другую, взяв за основу только что рассмотренные программы (см. листинги 9.3 и 9.4). К примеру, пусть имеется две битовые строки:

```
bit_str1    dd  0abcdefabh
bit_str2    dd  012345678h
```

Из этих строк получите строку

```
bit_str2   dd 0abcd34abh
```

Итоги

- Минимально адресуемая единица данных в процессоре — байт. Логические команды позволяют манипулировать отдельными битами. Только эти команды в системе команд процессора позволяют работать на битовом уровне. Этим, в частности, объясняется их важность.

* Работа на битовом уровне позволяет в отдельных случаях существенно сэкономить память, особенно при моделировании различных массивов, содержащих одноразрядные флаги или переключатели.
* Команды сдвига позволяют выполнять быстрое умножение и деление операндов на степени двойки, а также эффективное преобразование данных.
* Применение команд циклического сдвига и сдвига двойной точности позволяет реализовать максимально быстрые операции по рассогласованию, перемещению, вставке и извлечению битовых подстрок.

Глава 10

Команды передачи управления

- Программирование нелинейных алгоритмов
- Классификация команд передачи управления
- Команды безусловной передачи управления
- Понятие процедуры в языке ассемблера
- Команды условной передачи управления
- Средства организации циклов в языке ассемблера

В предыдущей главе мы познакомились с некоторыми командами, из которых формируются линейные участки программы. Каждая из них в общем случае выполняет некоторые действия по преобразованию или пересылке данных, после чего процессор передает управление следующей команде. Однако последовательно работают очень мало программ. Обычно в программе есть точки, в которых нужно принять решение о том, какая команда будет выполняться следующей. Это решение может быть:

- *безусловным* — в данной точке необходимо передать управление не следующей команде, а другой, которая находится на некотором удалении от текущей;
- *условным* — решение о том, какая команда будет выполняться следующей, принимается на основе анализа некоторых условий или данных.

Как вы помните, программа представляет собой последовательность команд и данных, занимающих определенное пространство оперативной памяти. Это пространство памяти может либо быть непрерывным, либо состоять из нескольких фрагментов. В главе 5 нами были рассмотрены средства сегментации кода программы

и ее данных. То, какая команда программы должна выполняться следующей, процессор узнает по содержимому пары регистров CS:(E)IP[1], в которой:

Si CS — регистр сегмента кода, в котором находится физический (базовый) адрес текущего сегмента кода;

ж EIP/IP — регистр указателя команды, в котором находится значение, представляющее собой смещение в памяти следующей выполняемой команды относительно начала текущего сегмента кода.

ПРИМЕЧАНИЕ Напомню, почему мы записываем регистры EIP/IP через косую черту. Какой конкретно регистр будет использоваться, зависит от установленных режимов адресации use16 или use32. Если указано use16, то используется IP, если use32 — то EIP.

Таким образом, команды передачи управления изменяют содержимое регистров CS и EIP, в результате чего процессор выбирает для выполнения не следующую по порядку команду программы, а команду в некотором другом участке программы. Конвейер внутри процессора при этом сбрасывается.

По принципу действия команды процессора, обеспечивающие организацию переходов в программе, можно разделить на три группы.

- Команды безусловной передачи управления:
  - безусловного перехода;
  - вызова процедуры и возврата из процедуры;
  - вызова программных прерываний и возврата из программных прерываний.
- Команды условной передачи управления:
  - перехода по результату команды сравнения;
  - перехода по состоянию определенного флага;
  - перехода по содержимому регистра ECX/CX.
- Команды управления циклом:
  - организации цикла со счетчиком ECX/CX;
  - организации цикла со счетчиком ECX/CX с возможностью досрочного выхода из цикла по дополнительному условию.

Возникает вопрос о том, каким образом обозначается то место, куда необходимо передать управление. В языке ассемблера это делается с помощью меток. *Метка* — это символическое имя, обозначающее определенную ячейку памяти и предназначенное для использования в качестве операнда в командах передачи управления.

Подобно переменной, транслятор ассемблера присваивает любой метке три атрибута:

[1] При обсуждении архитектуры микропроцессора мы говорили, что команды извлекаются из памяти заранее в так называемый конвейер, поэтому адрес подлежащей выборке команды из памяти и содержимое пары CS:E(IP) — не одно и то же. Эта пара регистров содержит адрес команды в программе, которая будет выполняться следующей, а не той команды, которая пойдет в конвейер.

- *имя сегмента кода,* где эта метка описана;
- *смещение* — расстояние в байтах от начала сегмента кода, в котором описана метка;
- *тип,* или *атрибут расстояния,* метки.

Последний атрибут может принимать два значения:

- near — переход на метку возможен только в пределах сегмента кода, где эта метка описана, то есть для перехода на метку физически достаточно изменить только содержимое регистра EIP/IP;
- far — переход на метку возможен только в результате межсегментной передачи управления, для осуществления которой требуется изменение содержимого как регистра EIP/IP, так и регистра CS.

Метку можно определить двумя способами:

- оператором : (двоеточие);
- директивой LABEL

Синтаксис первого способа показан на рис. 10.1.

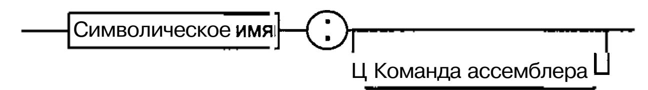


Рис. 10.1. Синтаксис описания метки с помощью оператора

С помощью оператора : можно определить метку только ближнего типа — near. Определенную таким образом метку можно использовать в качестве операнда в командах условных переходов JCC и безусловного перехода JMP, CALL Эти команды, естественно, должны быть в том же сегменте кода, в котором определена метка. Команда ассемблера может находиться как на одной строке с меткой, так и на следующей.

Во втором способе определения меток в программе используется директива LABEL (рис. 10.2).

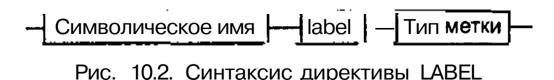


Рис. 10.2. Синтаксис директивы LABEL

Тип метки может иметь значение near или far. Обычно директиву LABEL используют для определения идентификатора заданного типа. Например, следующие описания меток ближнего типа эквивалентны:

```
ml:
    mov ax,pole_1

ml  label near
    mov  ax,pole_1
```

Понятно, что метка может быть только одного типа — либо near, либо far. Если возникает необходимость использовать для одной и той же команды метки и дальнего, и ближнего типов, то в этом случае необходимо определить две метки, причем метку дальнего типа нужно описать, используя директиву LABEL, как показано в следующем фрагменте:

```
;...
public    m_far     ;сделать метку m_far видимой для внешних программ
...
m_far     label  far ;определение метки дальнего типа m_far
m_near:              ;определение метки ближнего типа n_far
    movax,pole_1
...
```

Определив для команды mov ax,pole_1 две метки, можно организовывать переход на эту команду как из данного сегмента команд, так и из других сегментов команд, в том числе принадлежащих другим модулям. Для того чтобы сделать видимым извне имя метки m_far, применяется директива PUBLIC. К более подробному описанию этой директивы мы еще вернемся в главе 15.

Другой часто встречающийся случай применения директивы LABEL — это организация доступа к области памяти, содержащей данные разных типов, например:

```
...
mas_b   label   byte
mas_w   dw  15 dup (?)
...
;в этом фрагменте оба идентификатора относятся к одной области
;памяти и дают возможность работать с ней, используя разные
;имена, либо как с байтовым массивом, либо как с массивом слов
...
    mov mas_b+10,al  ;запись из al в массив байтов (в 11-й байт)
...
    mov mas_w,ax     ;запись из ах в первое слово области mas_w
...
```

Вспомним еще одно очень важное понятие ассемблера, имеющее прямое отношение к меткам, — *счетчик адреса команд.* Мы уже упоминали о нем в первых главах и говорили, что транслятор ассемблера обрабатывает исходную программу последовательно — команду за командой. При этом ведется счетчик адреса команд, который для первой исполняемой команды равен нулю, а далее, по ходу обработки очередной команды транслятором, увеличивается на длину этой команды. По сути, счетчик адреса команд — это смещение конкретной команды относительно начала сегмента кода. Таким образом, каждая команда во время трансляции имеет *адрес,* равный значению счетчика адреса команд. Обратитесь к главе 6 и еще раз посмотрите на приведенный в нем листинг 6.2. Первая колонка в листинге — номер строки файла листинга. Вторая колонка (или третья, если присутствует колонка с уровнем вложенности) — смещение команды относительно начала сегмента кода или, как мы сейчас определили, счетчик адреса. Значение, на которое он увеличивается по мере обработки ассемблером очередной строки исходной программы, равно значению длины машинной команды в этой строке. Исходя из этого, ясно, почему счетчик адреса увеличивается только после тех строк исходной программы, которые генерируют некоторое машинное представление (в том числе после директив резервирования и инициализации данных в сегменте данных).

Транслятор ассемблера обеспечивает нам две возможности работы с этим счетчиком:

- использование меток, атрибуту смещения которых транслятор присваивает значение счетчика адреса следующей команды;
- применение для обозначения счетчика адреса команд при задании операндов команд специального символа $, во время трансляции заменяемого текущим численным значением счетчика адреса.

Классический пример:

```
...
.data
;вычисление длины строки в сегменте данных
Str_Mes db  "Займись делом - учи ассемблер ..."
Len_Msg=$-Str_Mes
...
```

После ассемблирования значение Len_Msg будет равно длине строки, так как значение символа $ в месте его появления отличается от Str_Mes ровно на длину строки.

Другое применение, избавляющее программиста от необходимости «плодить» лишние метки в программе, — реализация близкого, буквально через следующую или предыдущую команды, перехода. Для примера рассмотрим фрагмент программы из листинга 6.1 (см. главу 6):

```
...
    cmp dl,9h       ;сравнить (dl) с 9h
    jle $+5         ;перейти на команду mov cl,4h, если dl<9h или dl=9h
    sub dl,7h       ;вычитание: (dl)=(dl)Ч7h
;M1:
    mov cl,4h       ;пересылка 4h в регистр cl
...
    cmp al,9h       ;сравнить (al) с 9h 28
    jle  $+4        ;перейти на команду add dl,al, если al<9h или al=9h
    sub al,7h       ;вычитание: (al)=(al)Ч7h
;M2:
    add dl,al
...
```

Как узнать правильные значения длины команд? Во-первых, по опыту, то.есть просто догадаться. Во-вторых, из файла листинга, что не всегда дает результат, так как конечные машинные команды (колонка файла листинга с объектным кодом) в нем не всегда до конца сформированы. В-третьих, узнать длины команд можно, загрузив программу в отладчик и активировав окно CPU. Последний способ самый точный. Не будет ничего страшного, если для первого прогона исполняемого модуля программы в нем будут неверные значения для подобных относительных переходов. И еще одно замечание: при переходе вперед необходимо учитывать длину текущей команды перехода, при переходе назад этого делать не нужно.

Кроме возможности получения значения счетчика адреса компилятор позволяет при необходимости установить счетчик адреса в нужное абсолютное значение. Это делается с помощью директивы ORG:

ORG выражение

Здесь выражение должно быть таким, чтобы ассемблер мог преобразовать его к абсолютному числу при первом проходе трансляции.

К примеру, эту директиву всегда используют при создании исполняемого файла с расширением .com. В контексте нашего обсуждения поясним, в чем здесь суть. В главе 5 мы обсуждали сегментацию и деление программы на сегменты. Программа в формате COM состоит из одного сегмента величиной не более 64 Кбайт. Сегментные регистры CS и DS содержат одно и то же значение физического адреса, а регистр SS указывает на конец этого единственного сегмента. Программа-загрузчик операционной системы, считывая с диска исполняемые файлы с расширениями .exe и .com, производит определенные действия. В частности, настраивает переме-

щаемые адреса программ на их конкретные физические значения. Кроме того, к началу каждой исполняемой программы в памяти добавляется специальная область величиной 256 байт (100h) — *префикс программного сегмента* (PSP). Он предназначен для хранения различной информации о загруженном исполняемом модуле. Для программ формата COM блок PSP находится в начале сегмента размером в 64 Кбайт. В исходной программе, для исполняемого файла которой планируется формат COM, мы должны предусмотреть место для блока PSP, что и делается директивой org 100h.

Чтобы закончить разговор о файлах этого типа, разберемся с тем, как получить исполняемый модуль формата COM. Трансляция программы выполняется как обычно. Далее возможны два варианта действий.

- Во-первых, возможно использование утилиту tlink с ключом /t:

  ```
  tlink /t имя_объектного_файла
  ```

  Этот вариант подходит только в том случае, если вы правильно оформили исходный текст программы для формата COM. Кроме того, наличие директивы org 100h предполагает:

  - отсутствие разделения сегментов данных и стека, то есть данные необходимо описать в сегменте кода, а для их обхода использовать команду безусловного перехода JMP;
  - применение директивы ASSUME для указания транслятору на необходимость связать содержимое регистров DS и SS с сегментом кода:

  ```
  codeseg segment para "code"
      assume  cs: codeseg,ds:codeseg,ss:codeseg
      org 100h
      jmp ml
  ;здесь описываем данные
  ml:
  ;далее идут команды программы
  ...
  ```

- Во-вторых, можно использовать специальную утилиту exe2bin. Эта утилита позволяет преобразовать уже полученный ранее исполняемый модуль в формате EXE в формат COM:

  ```
  exe2bin имя_файла_exe имя_файла_com.com
  ```

  Этот вариант не требует специального оформления исходного текста программы. Единственным требованием является то, чтобы исходный текст был довольно мал по объему.

Интересный вариант использования директивы ORG рассмотрен в [8], где с ее помощью выполняется динамическая модификация машинного кода команды.

Безусловные переходы

Предыдущее обсуждение выявило некоторые детали механизма перехода. Команды перехода модифицируют регистр указателя команды EIP/IP и, возможно, сегментный регистр кода CS. Что именно должно подвергнуться модификации, зависит:

- от типа операнда в команде безусловного перехода (ближний или дальний);
- от модификатора, который указывается перед адресом перехода в команде перехода и может принимать следующие значения (сам адрес при прямом переходе находится непосредственно в команде, а при косвенном — в регистре или ячейке памяти):
  - NEAR PTR — прямой переход на метку внутри текущего сегмента кода, при этом модифицируется только регистр EIP/IP (в зависимости от заданного типа сегмента кода use16 или use32) на основе указанного в команде адреса (метки) или выражения, использующего символ извлечения значения счетчика адреса команд ($);
  - FAR PTR — прямой переход на метку в другом сегменте кода, при этом адрес перехода задается в виде непосредственного операнда или адреса (метки) и состоит из 16-разрядного селектора и 16/32-разрядного смещения, которые загружаются, соответственно, в регистры CS и EIP/IP;
  - WORD PTR — косвенный переход на метку внутри текущего сегмента кода, при этом модифицируется (значением смещения размером 16 или 32 бита из памяти по указанному в команде адресу или из регистра) только регистр EIP/IP;
  - DWORD PTR — косвенный переход на метку в другом сегменте кода, при этом модифицируются (значением из памяти — и только из памяти, из регистра нельзя) оба регистра, CS и EIP/IP (первое слово/двойное слово адреса перехода, представляющее собой смещение, загружается в EIP/IP; второе/третье слово — в CS).

Команда безусловного перехода

Синтаксис команды безусловного перехода без сохранения информации о точке возврата:

```
jmp [модификатор] адрес_перехода
```

Здесь адрес_перехода представляет метку или адрес области памяти, в которой находится указатель перехода.

Всего в системе команд процессора есть несколько кодов машинных команд безусловного перехода JMP. Их различия определяются дальностью перехода и способом задания целевого адреса. Дальность перехода определяется местоположением операнда адрес_перехода. Этот адрес может находиться в текущем сегменте кода или в некотором другом сегменте. В первом случае переход называется *внутрисегментным,* или *близким,* во втором — *межсегментным,* или *дальним.*

Внутрисегментный переход предполагает, что изменяется только содержимое регистра EIP/IP. Можно выделить три варианта внутрисегментного использования команды JMP:

- прямой короткий переход;
- прямой переход;
- косвенный переход.

Прямой короткий внутрисегментный переход применяется, когда расстояние от команды JMP до адреса перехода не превышает –128 или +127 байт. В этом случае транслятор ассемблера формирует машинную команду безусловного перехода длиной всего два байта (размер обычной команды внутрисегментного безусловного перехода составляет три байта). Первый байт в этой команде — код операции, значение которого говорит о том, что процессор должен особым образом трактовать второй байт команды. Значение второго байта вычисляется транслятором как разность между значением смещения команды, следующей за JMP, и значением адреса перехода. При осуществлении прямого короткого перехода нужно иметь в виду следующий важный момент, связанный с местоположением адреса перехода и самой команды JMP. Если адрес перехода расположен до команды JMP, то ассемблер формирует короткую команду безусловного перехода без дополнительных указаний. Если адрес перехода расположен после команды JMP, транслятор не может сам определить, что переход короткий, так как у него еще нет информации об адресе перехода. Для оказания помощи компилятору в формировании команды короткого безусловного перехода в дополнение к ранее упомянутым модификаторам используют модификатор SHORT PTR:

```
...
    jmp short ptr ml
... ;не более 35-40 команд (127 байт)
m1:
```

Еще вариант:

```
...
ml:
...;не более 35-40 команд (-128 байт)
    jmp ml
...
```

Прямой внутрисегментный переход отличается от прямого короткого внутрисегментного перехода тем, что длина машинной команды JMP в этом случае составляет три байта. Увеличение длины связано с тем, что поле адреса перехода в машинной команде JMP расширяется до двух байтов, а это, в свою очередь, позволяет производить переходы в пределах 64 Кбайт относительно следующей за JMP команды:

```
m1:
...;расстояние более 128 байт и менее 64 Кбайт
    jmp mt
...
```

Косвенный внутрисегментный переход подразумевает «косвенность» задания адреса перехода. Это означает, что в команде указывается не сам адрес перехода, а место, где он «лежит». Приведем несколько примеров, в которых двухбайтовый адрес перехода выбирается либо из регистра, либо из области памяти:

```
    lea bx,m1
    jmp bx  ;адрес перехода в регистре bx
...
ml:
...
.data
addr_m1 dw ml
...
.code
```

```
;...
   jmp addr_m1          ;адрес перехода в ячейке памяти addr_m1
;...
m1:
```

Еще несколько вариантов косвенного внутрисегментного перехода:

```
<1>...
<2>.data
<3>addr dw   ml
<4> dw   m2
<5>...
<6>.code
<7>...
<8>cycl:
<9>     mov si,0
<10     jmp addr[si];адрес перехода в слове памяти addr+(si)
<11>...
<12>    mov  si,2
<13>    jmp  cycl
<14>m1:
<15>...
<16>m2:
<17>...
```

В этом примере одна команда JMP (строка 10) может выполнять переходы на разные метки. Выбор конкретной метки перехода определяется содержимым регистра SI. Операнд команды JMP определяет адрес перехода косвенно после вычисления выражения addr+(SI).

```
<1>...
<2>.data
<3>addr dw  ml
<4>...
<5>.code
<6>...
<7>     lea  si,addr
<8>    jmp near ptr[si] ;адрес перехода в ячейке памяти addr
<9>...
<10>m1:
```

В данном случае указание модификатора NEAR PTR обязательно, так как, в отличие от предыдущего способа, адрес ячейки памяти addr с адресом перехода транслятору передается неявно (строки 3, 7 и 8), и, не имея информации о метке, он не может определить, какой именно переход осуществляется — внутрисегментный или межсегментный. *Межсегментный переход* предполагает другой формат машинной команды JMP. При осуществлении межсегментного перехода кроме регистра EIP/IP модифицируется также регистр CS. Аналогично внутрисегментному переходу, межсегментный переход поддерживают два варианта команд безусловного перехода: прямой и косвенный.

Команда *прямого межсегментного перехода* имеет длину пять байтов, из которых два байта составляют значение смещения и два байта — значение сегментной составляющей адреса:

```
seg_l   segment
;...
   jmp  far  ptr  m2   ;здесь far обязательно
;...
ml  label   far
;...
seg_l   ends
```

```
seg_2   segment
;...
m2  label   far
    jmp ml  ;здесь far необязательно
```

Рассматривая этот пример, обратите внимание на модификатор FAR PTR в команде JMP. Его необходимость объясняется все той же логикой работы однопроходного транслятора. Если описание метки (метка ml) встречается в исходном тексте программы раньше, чем соответствующая ей команда перехода, то задание модификатора необязательно, так как транслятор все знает о данной метке и сам формирует нужную пятибайтовую форму команды безусловного перехода. В случае, когда команда перехода встречается до описания соответствующей метки, транслятор не имеет еще никакой информации о метке, и модификатор FAR PTR в команде JMP опускать нельзя, так как транслятор не знает, какую форму команды формировать — трехбайтную или пятибайтную. Без специального указания модификатора транслятор будет формировать трехбайтную команду внутрисегментного перехода.

Команда *косвенного межсегментного перехода* в качестве операнда имеет адрес области памяти, в которой содержатся смещение и сегментная часть целевого адреса перехода:

```
data    segment
addr_m1 dd  ml  ;в поле addr_m1 значения смещения
                ;и адреса сегмента метки ml
data    ends
code_l  segment
;...
    jmp ml
;...
code_l  ends
code_2  segment
;...
ml  label   far
    mov ax,bx
;...
code_2  ends
```

Как вариант косвенного межсегментного перехода необходимо отметить *косвенный регистровый межсегментный переход.* В этом виде перехода адрес перехода указывается косвенно — в регистре. Это очень удобно для программирования динамических переходов, в которых адрес перехода может изменяться на стадии выполнения программы:

```
data    segment
addr_ml dd  ml  ;в поле addr_m1 значения смещения
                ;и адреса сегмента метки ml
data    ends
code_l  segment
;...
    lea bx,addr_m1
    jmp dword ptr[bx]
;...
code_l  ends
code_2  segment
;...
ml  label   far
    mov ax,bx
;...
code_2  ends
```

Таким образом, модификаторы SHORT PTR, NEAR PTR и WORD PTR применяются для организации внутрисегментных переходов, а FAR PTR и DWORD PTR — межсегментных.

Для полной ясности нужно еще раз подчеркнуть, что если тип сегмента — use32, то в тех местах, где речь шла о регистре IP, можно использовать регистр EIP и, соответственно, размеры полей смещения увеличить до 32 битов.

Процедуры

До сих пор мы рассматривали примеры программ, предназначенные для однократного выполнения. Но, приступив к программированию достаточно серьезной задачи, вы наверняка столкнетесь с тем, что у вас появятся повторяющиеся фрагменты кода. Одни из них могут состоять всего из нескольких команд, другие занимать и достаточно много места в исходном коде. В последнем случае эти фрагменты существенно затруднят чтение текста программы, снизят ее наглядность, усложнят отладку и послужат неисчерпаемым источником ошибок. В языке ассемблера есть несколько средств, решающих проблему дублирования фрагментов программного кода. К ним относятся:

- процедуры;
- макроподстановки (макроассемблер);
- генерация и обработка программных прерываний.

В данной главе рассматриваются только основные понятия, относящиеся к вызову процедур. Ввиду важности этого вопроса мы продолжим его изучение в главе 15 в контексте темы модульного программирования на ассемблере. Актуальная для программирования под Windows проблема разработки библиотек DLL на ассемблере описана в [8]. Макроассемблеру посвящена глава 14.

Процедура, или *подпрограмма,* — это основная функциональная единица декомпозиции (разделения на части) некоторой задачи. Процедура представляет собой группу команд для решения конкретной подзадачи и обладает средствами получения управления из точки вызова задачи более высокого уровня и возврата управления в эту точку. В простейшем случае программа может состоять из одной процедуры. Другими словами, процедуру можно определить как правильным образом оформленную совокупность команд, которая, будучи однократно описана, при необходимости может быть вызвана в любом месте программы.

Для описания последовательности команд в виде процедуры в языке ассемблера используются две директивы: PROC и ENDP.

Синтаксис описания процедуры таков (рис. 10.3).

Из рисунка видно, что в заголовке процедуры (директиве PROC) обязательным является только задание имени процедуры. Среди большого количества операндов директивы PROC следует особо выделить [расстояние]. Этот атрибут может принимать значения NEAR или FAR и характеризует возможность обращения к процедуре из другого сегмента кода. По умолчанию атрибут [расстояние] принимает значение NEAR.

Процедура может размещаться в любом месте программы, но так, чтобы на нее случайным образом не попало управление. Если процедуру просто вставить в общий поток команд, то процессор воспримет команды процедуры как часть этого

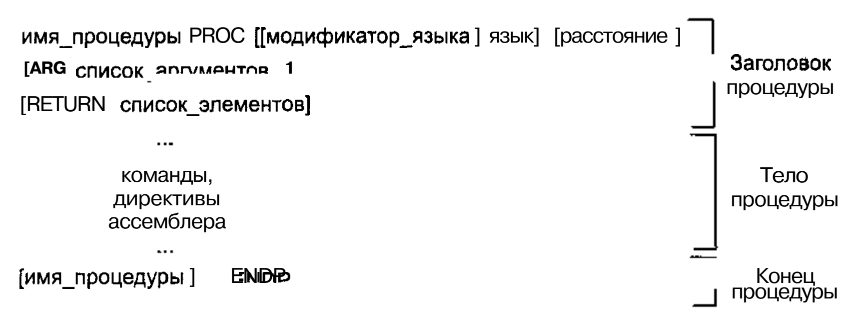


Рис. **10.3.** Синтаксис описания процедуры в программе

потока и, соответственно, начнет выполнять эти команды. Учитывая это обстоятельство, есть следующие варианты размещения процедуры в программе:

- в начале программы (до первой исполняемой команды);
- в конце программы (после команды, возвращающей управление операционной системе);
- промежуточный вариант — внутри другой процедуры или основной программы (в этом случае необходимо предусмотреть обход процедуры с помощью команды безусловного перехода JMP);
- в другом модуле (библиотеке DLL).

Размещение процедуры в начале сегмента кода предполагает, что последовательность команд, ограниченная парой директив PROC и ENDP, будет размещена до метки, обозначающей первую команду, с которой начинается выполнение программы. Эта метка должна быть указана как параметр директивы END, обозначающей конец программы:

```
model   small
.stack  100h
.data
.code
my_proc procnear
...
ret
my_proc endp
start:
...
end start
```

Объявление имени процедуры в программе равнозначно объявлению метки, поэтому директиву PROC в частном случае можно рассматривать как завуалированную форму определения программной метки. Поэтому сама исполняемая программа также может быть оформлена в виде процедуры, что довольно часто и делается с целью пометить первую команду программы, с которой должно начаться выполнение. При этом не забывайте, что имя этой процедуры нужно обязательно указывать в заключительной директиве END. Такой синтаксис мы уже неоднократно использовали в своих программах. Так, последний рассмотренный фрагмент эквивалентен следующему:

```
model   small
.stack  100h
```

```
.data
.code
my_proc  procnear
...
ret
my_proc endp
start    proc
...
start    endp
end start
```

В этом фрагменте после загрузки программы в память управление будет передано первой команде процедуры с именем start.

Размещение процедуры *в конце программы* предполагает, что последовательность команд, ограниченная директивами PROC и ENDP, находится следом за командой, возвращающей управление операционной системе:

```
model   small
.stack  100h
.data
.code
start:
...
    mov ax,4c00h
    int 21h ;возврат управления операционной системе
my_proc procnear
...
ret
my_proc endp
end start
```

Промежуточный вариант расположения тела процедуры предполагает ее размещение внутри другой процедуры или основной программы. В этом случае необходимо предусмотреть обход тела процедуры, ограниченного директивами PROC и ENDP, с помощью команды безусловного перехода ЗМР:

```
model   small
.stack  100h
.data
.code
start:
...
    jmp ml
my_proc  procnear
...
ret
my_proc endp
ml:
...
    mov ax,4c00h
    int 21h  ;возврат управления операционной системе
end start
```

Последний вариант расположения описаний процедур — *в отдельном сегменте кода* — предполагает, что часто используемые процедуры выносятся в отдельный файл, который должен быть оформлен как обычный исходный файл и подвергнут трансляции для получения объектного кода. Впоследствии этот объектный файл с помощью утилиты tlink можно объединить с файлом, в котором данные процедуры используются. С утилитой tlink мы познакомились в главе 6. Этот способ предполагает наличие в исходном тексте программы еще некоторых элементов, связанных с особенностями реализации концепции модульного программирования

в языке ассемблера. Поэтому в полном объеме этот способ будет рассмотрен в главе 15.

Как обратиться к процедуре? Так как имя процедуры обладает теми же атрибутами, что и обычная метка в команде перехода, то обратиться к процедуре, в принципе, можно с помощью любой команды перехода. Но есть одно важное свойство, которое можно использовать благодаря специальному *механизму вызова процедур*. Суть состоит в возможности сохранения информации о контексте программы в точке вызова процедуры. Под *контекстом* понимается информация о состоянии программы в точке вызова процедуры. В системе команд процессора есть две команды для работы с контекстом — CALL и RET.

- Команда CALL осуществляет вызов процедуры (подпрограммы). Синтаксис команды:

  ```
  call [модификатор] имя_процедуры
  ```

  Подобно команде JMP команда CALL передает управление по адресу с символическим именем **имя_процедуры**, но при этом в стеке сохраняется адрес возврата (то есть адрес команды, следующей после команды CALL).

- Команда RET считывает адрес возврата из стека и загружает его в регистры CS и EIP/IP, тем самым возвращая управление на команду, следующую в программе за командой CALL Синтаксис команды:

  ```
  ret [число]
  ```

  Необязательный параметр [число] обозначает количество элементов, удаляемых из стека при возврате из процедуры. Размер элемента определяется хорошо знакомыми нам параметрами директивы SEGMENT — **use16** и **use32** (или соответствующим параметром упрощенных директив сегментации). Если указан параметр use16, то [число] — это значение в байтах; если use32 — в словах.

Для команды CALL, как и для JMP, актуальна проблема организации ближних и дальних переходов. Это видно из формата команды, где присутствует параметр [модификатор]. Как и в случае команды JMP, вызов процедуры командой CALL может быть внутрисегментным и межсегментным.

- При *внутрисегментном* вызове процедура находится в текущем сегменте кода (имеет тип near), и в качестве адреса возврата команда CALL сохраняет только содержимое регистра IP/EIP, что вполне достаточно (рис. 10.4).
- При *межсегментном* вызове процедура находится в другом сегменте кода (имеет тип far), и для осуществления возврата команда CALL должна запомнить содержимое обоих регистров (CS и IP/EIP), при этом в стеке сначала запоминается содержимое регистра CS, затем — регистра IP/EIP (рис. 10.5).

Важно отметить, что одна и та же процедура не может быть одновременно процедурой ближнего и дальнего типов. Таким образом, если процедура используется в текущем сегменте кода, но может вызываться и из другого сегмента программы, то она должна быть объявлена процедурой типа far. Подобно команде JMP, существуют четыре разновидности команды CALL. Какая именно команда будет сформирована, зависит от значения *модификатора* в команде вызова процедуры CALL и атрибута дальности в описании процедуры. Если процедура описана в начале сегмента данных с указанием дальности в ее заголовке, то при ее вызове параметр

Процедура ближнего типа my_proc:

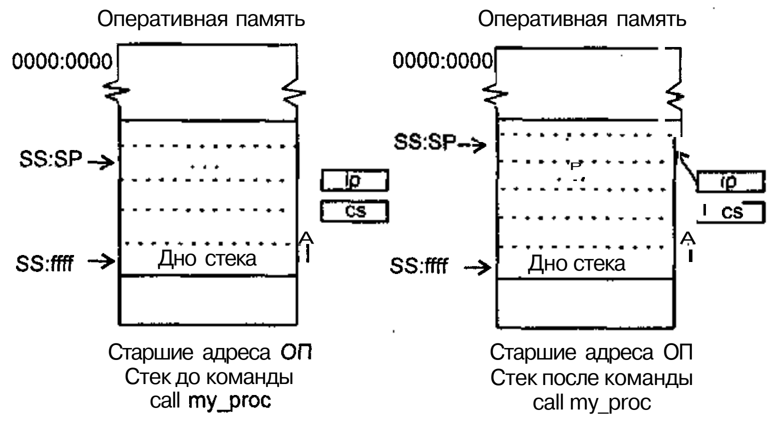


Рис. 10.4. Содержимое стека до и после выполнения команды вызова процедуры ближнего типа

Процедура дальнего типа my_proc:

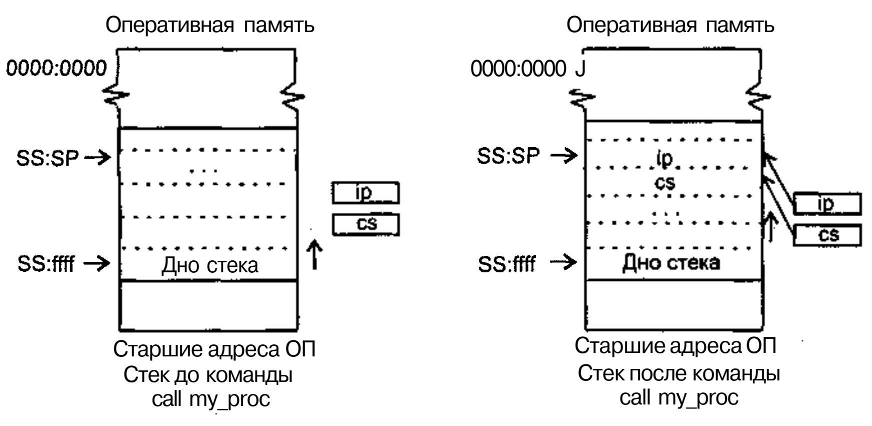


Рис. 10.5. Содержимое стека до и после выполнения команды вызова процедуры дальнего типа

[модификатор] можно не указывать: транслятор сам разберется, какую команду CALL ему нужно сформировать. Если же процедура описана после ее вызова, например, в конце текущего сегмента или в другом сегменте, то при ее вызове нужно указать ассемблеру тип вызова, чтобы он мог за один проход правильно сформировать команду CALL. Значения *модификатора* такие же, как и у команды ЗМР, за исключением значения SHORT PTR.

С директивой PROC используются еще несколько директив: ARG, RETURNS, LOCAL, USES. Их назначение — помочь программисту выполнить некоторые рутинные действия при вызове и возврате из процедуры (заодно и повысив надежность кода). Директивы ARG и RETURNS назначают входным и выходным параметрам процедуры, передаваемым через стек, символические имена. Директива USES в качестве параметров содержит имена используемых в процедуре регистров. При обработке

этой директивы ассемблер формирует входной и выходной коды процедуры (из команд PUSH и POP), обеспечивающие сохранение и восстановление регистров. Директива LOCAL предназначена для выделения кадра стека для локальных переменных, что позволяет экономить память, занимаемую программой в целом. Подробно эти директивы обсуждаются в главе 15.

Необходимо заметить, что в данном разделе приведена информация о порядке описания процедур, принятом в TASM. Описание и использование процедур в MASM имеет особенности, о которых можно узнать из материала главы 15.

Последний и, наверное, самый важный вопрос, возникающий при работе с процедурами, — как правильно передать параметры процедуре и вернуть результат? Этот вопрос тесно связан с концепцией модульного программирования и подробно будет рассматриваться в главе 15. С примерами использования процедур вы можете познакомиться в листингах подпрограмм, предназначенных для вычисления четырех основных арифметических действий с двоичными и десятичными (BCD) числами и находящихся среди прилагаемых к книге файлов в каталоге главы 8[1]. Кроме того, вопросы организации рекурсивных и вложенных процедур рассмотрены в [8].

Условные переходы

До сих пор мы рассматривали команды перехода с «безусловным» принципом действия, но в системе команд процессора есть большая группа команд, призванных самостоятельно принимать решение о том, какая команда должна выполняться следующей. Решение принимается в зависимости от определенных условий, определяемых конкретной командой перехода. Процессор поддерживает 18 команд условного перехода, позволяющих проверить:

- отношение между операндами со знаком (больше или меньше);
- отношение между операндами без знака (выше или ниже)[2];
- состояниями арифметических флагов ZF, SF, CF, OF, PF (но не AF).

Команды условного перехода имеют одинаковый синтаксис:

```
jcc метка_перехода
```

Как видно, мнемокод всех команд начинается с символа «j» — от *слова jump* (прыжок). Вместо символов «cc» указывается конкретное условие, анализируемое командой. Что касается операнда метка_перехода, то он определяет метку перехода, которая может находиться только в пределах текущего сегмента кода; межсегментной передачи управления в условных переходах не допускается. В связи с этим отпадает вопрос о модификаторе, который присутствовал в синтаксисе команд безусловного перехода. В ранних моделях процессора (8086, 80186 и 80286)

[1] Все прилагаемые к книге файлы можно найти по адресу http://www.piter.com/download. — *Примеч. ред.*

[2] Термины «больше или меньше» и «выше или ниже» происходят от соответствующих английских терминов «greater-less» и «above-below». Первые буквы этих терминов входят в состав мнемонических обозначений соответствующих команд условного перехода. Несмотря на кажущуюся синонимичность этих терминов, на самом деле они отражают тот факт, что соответствующие им команды условного перехода анализируют разные флаги (см. также далее пояснения в тексте).

команды условного перехода могли осуществлять только короткие переходы — на расстояние от -128 до +127 байт от команды, следующей за командой условного перехода. Начиная с процессора 80386 это ограничение снято, но, как видите, только в пределах текущего сегмента кода.

Для того чтобы принять решение о том, куда будет передано управление командой условного перехода, предварительно должно быть сформировано условие, на основании которого должно приниматься решение. Источниками такого условия могут быть:

- любая команда, изменяющая состояние арифметических флагов;
- команда CMP, сравнивающая значения двух операндов;
- состояние регистра ECX/CX.

Обсудим эти варианты, чтобы разобраться с тем, как работают команды условного перехода.

Команда сравнения

Команда сравнения CMP (*CoMPare*) имеет интересный принцип работы. Он абсолютно такой же, как у команды вычитания SUB (см. главу 8). Команда CMP так же, как и команда SUB, выполняет вычитание операндов и по результатам сравнения устанавливает флаги. Единственное, чего она не делает, — не записывает результат вычитания на место первого операнда.

Синтаксис команды CMP:

```
cmp операнд_1,операнд_2
```

Флаги, устанавливаемые командой CMP, можно анализировать специальными командами условного перехода. Прежде чем мы их рассмотрим, уделим немного внимания мнемонике этих команд (табл. 10.1). Понимание обозначений (элементов в названии команды JCC, обозначенных нами символами «cc») при формировании названия команд условного перехода облегчит их запоминание и дальнейшее практическое использование.

Таблица 10.1. Значение аббревиатур в названии команды jcc

Мнемоническое обозначение	Оригинальный термин	Перевод	Тип операндов
e	Equal	Равно	Любые
n	Not	Нет	Любые
g	Greater	Больше	Числа со знаком
l	Less	Меньше	Числа со знаком
a	Above	Выше (в смысле больше)	Числа без знака
b	Below	Ниже (в смысле меньше)	Числа без знака

В табл. 10.2 представлен перечень команд условного перехода для команды CMP. Не удивляйтесь тому обстоятельству, что одинаковым значениям флагов соответствуют несколько разных мнемокодов команд условного перехода (они отделены друг от друга косой чертой). Разница в названии обусловлена желанием разработ-

чиков процессора упростить использование команд условного перехода с разными группами команд. Поэтому разные названия отражают скорее разную функциональную направленность. Тем не менее, то, что эти команды реагируют на одни и те же флаги, делает их абсолютно эквивалентными и равноправными в программе. Именно поэтому они сгруппированы не по названиям, а по значениям флагов (условиям), на которые они реагируют.

Таблица 10.2. Перечень команд условного перехода для команды cmp

Типы операндов	Мнемокод команды условного перехода	Критерий условного перехода	Значения флагов для перехода
Любые	JE	операнд_1 = операнд_2	ZF = 1
Любые	JNE	операнд_1 <> операнд_2	ZF = 0
Со знаком	JL/JNGE	операнд_1 < операнд_2	SF <> OF
Со знаком	JLE/JNG	операнд_1 <= операнд_2	SF <> OF или ZF = 1
Со знаком	JG/JNLE	операнд_1 > операнд_2	SF = OF и ZF = 0
Со знаком	JGE/JNL	операнд_1 => операнд_2	SF = OF
Без знака	JB/JNAE	операнд_1 < операнд_2	CF = 1
Без знака	JBE/JNA	операнд_1 <= операнд_2	CF = 1 или ZF=1
Без знака	JA/JNBE	операнд_1 > операнд_2	CF - 0 и ZF - 0
Без знака	JAE/JNB	операнд_1 => операнд_2	CF = 0

В качестве примера применения команды CMP рассмотрим фрагмент программы, который обнуляет поле pole_m длиной *n* байт:

```
.data
n     equ  50
pole_m  db  n dup (?)
.code
;...
      xor  bx,bx          ;bx=0
ml:   mov  mem[bx],0
      inc  bx
      cmp  bx,n
      jne  ml
exit:
;...
```

Так как команды условного перехода не изменяют флагов, то после одной команды CMP вполне могут следовать несколько команд условного перехода. Это может быть сделано для того, например, чтобы исследовать каждую из альтернативных ветвей: больше, меньше или равно:

```
.data
mas db  dup (?)
.code
;...
      cmp mas[si],5    ;сравнить очередной элемент массива с 5
      je  eql          ;переход, если элемент mas равен 5
      jl  low          ;переход, если элемент mas меньше 5
      jg  grt          ;переход, если элемент mas больше 5
eql:
```

```
    ...
low:
    ...
grt:
    ...
```

Команды условного перехода и флаги

Мнемоническое обозначение некоторых команд условного перехода отражает название флага, с которым они работают, и имеет следующую структуру: первым идет символ «j» (jump — переход), вторым — либо обозначение флага, либо символ отрицания «n», после которого стоит название флага. Такая структура команды отражает ее назначение. Если символа «n» нет, то проверяется состояние флага и, если он равен 1, производится переход на метку перехода. Если символ «n» присутствует, то проверяется состояние флага на равенство 0 и в случае успеха производится переход на метку перехода. Мнемокоды команд, названия флагов и условия переходов приведены в табл. 10.3. Эти команды можно использовать после любых команд, изменяющих указанные флаги.

Таблица 10.3. Команды условного перехода и флаги

Название флага	Номер бита в регистре **eflags/flags**	Команда условного перехода	Значение флага для осуществления перехода
Переноса CF	1	JC	CF = 1
Четности PF	2	JP	PF = 1
Нуля ZF	6	JZ	ZF = 1
Знака SF	7	JS	SF = 1
Переполнения OF	11	JO	OF = 1
Переноса CF	1	JNC	CF = 0
Четности PF	2	JNP	PF = 0
Нуля ZF	6	JNZ	ZF = 0
Знака SF	7	jns	sf = 0
Переполнения OF	11	JNO	OF = 0

Если внимательно посмотреть на табл. 10.2 и 10.3, видно, что многие команды условного перехода в них эквивалентны, так как в них анализируются одинаковые флаги.

В листинге 10.1 приведен пример программы, производящей в строке символов длиной *n* байт замену строчных букв английского алфавита прописными. Для осмысленного рассмотрения этого примера вспомним ASCII-коды, соответствующие этим буквам (см. главу 6). Строчные и прописные буквы в таблице ASCII упорядочены по алфавиту. Строчным буквам соответствует диапазон кодов 61h-7ah, прописным — 41h–5ah. Для того чтобы понять идею, лежащую в основе алгоритма преобразования, достаточно сравнить представления соответствующих прописных и строчных букв в двоичном виде:

a — 0110 0001...z — 0111 1010

A - 0100 0001...Z - 0101 1010

Как видно из приведенного двоичного представления, для выполнения преобразования между строчными и прописными буквами достаточно всего лишь инвертировать 5-й бит.

Листинг 10.1. Смена регистра символов

```
<1> ;prg_10_1.asm
<2> model   small
<3> .stack  100h
<4> .data
<5> n   equ 10                 ;количество символов в stroka
<6> stroka  db   "acvfgrndup"
<7> .code
<8> start:
<9>      mov ax,@data
<10>         mov ds,ax
<11>         xor ax,ax
<12>         mov cx,n
<13>         lea bx,stroka    ;адрес stroka в bx
<14>     ml: mov al,[bx]      ;очередной символ из stroka в al
<15>         cmp al,61h       ;убедиться, что код символа не меньше 61h
<16>         jb  next         ;если меньше, то не обрабатывать
<17>                          ;и перейти к следующему символу
<18>         cmp al,7ah       ;убедиться, что код символа не больше 7ah
<19>         ja  next         ;если больше, то не обрабатывать
<20>                          ;и перейти к следующему символу
<21>         and al,11011111b;инвертировать 5-й бит
<22>         mov  [bx],al     ;символ - на его место в stroka
<23>     next:
<24>         inc bx           ;адресовать следующий символ
<25>         dec cx           ;уменьшить значение счетчика в cx
<26>         jnz ml           ;если cx не 0, то переход на ml
<27>     exit:
<28>         mov ax,4c00h
<29>         int 21h          ;возврат управления операционной системе
<30>     end start
```

Обратите внимание на строку 25 листинга. Команда DEC уменьшает значение регистра CX на 1. Когда это значение станет равным 0, процессор по результату операции декремента установит флаг ZF. Команда в строке 26 анализирует состояние этого флага и, пока он не равен 1 (см. табл. 10.3), передает управление на метку ml. Заметьте, что на место этой команды можно было бы поставить команду JNE (см. табл. 10.2). Но для анализа регистра CX в системе команд процессора есть специальная команда, которую мы сейчас и рассмотрим.

Команды условного перехода и регистр ECX/CX

Архитектура процессора предполагает специфическое использование многих регистров. К примеру, регистр EAX/AX/AL используется как аккумулятор, а регистры EBP/BP, ESP/SP — для работы со стеком. Регистр ECX/CX тоже имеет определенное функциональное назначение — он выполняет функции счетчика в командах управления циклами и при работе с цепочками символов. Возможно, что функционально команды условного перехода, связанные с регистром ECX/CX, правильнее было бы отнести к этой группе команд.

Синтаксис команд JCXZ (*Jump* if cx is Zero — переход, если CX ноль) и JECXZ (*Jump* Equal ecx Zero — переход, если ECX ноль) таков:

```
jcxz/jecxz метка_перехода
```

Эти команды очень удобно использовать при организации цикла и при работе с цепочками символов. В этой главе мы разберемся со средствами организации циклов в программах на языке ассемблера и покажем работу команд JCXZ/JECXZ. Глава 12 будет посвящена цепочечным командам, где мы еще раз вернемся к командам JCXZ/JECXZ. Нужно отметить свойственное им ограничение. В отличие от других команд условной передачи управления, команды JCXZ/JECXZ могут адресовать только короткие переходы — на -128 байт или на +127 байт от следующей за ней команды.

Установка байта по условию

При рассмотрении команд условного перехода логично упомянуть еще об одной команде — SETcc. Данная команда впервые появилась в процессоре i386. Ее формат:

```
SETcc операнд1
```

Команда устанавливает байт операнд1 после проверки модификатора cc, задающего условие, аналогично тому, как это делается в командах условного перехода. Фактически, модификатор cc обозначает флаг, который нужно проверить (см. приложение). В качестве операнда используется либо один из 8-разрядных регистров (AL, AH, BL, BH, CL, CH, DL, DH), либо адрес ячейки памяти размером в байт. Алгоритм работы команды заключается в том, что значение операнда устанавливается по результатам проверки условия, заданного модификатором cc:

в операнд1 = 0 — если условие ложно;

я операнд1 = 1 — если условие истинно.

К примеру, в следующем фрагменте байт по адресу rez будет установлен в 0:

```
mov ax,0
cmp ax,1
sete   byte ptr rez
```

Организация циклов

Цикл, как известно, представляет собой важную алгоритмическую структуру, без которой не обходится, наверное, ни одна программа. Организовать циклическое выполнение некоторого фрагмента программы можно, к примеру, используя команды условной передачи управления или команду безусловного перехода JMP. Например, подсчитаем количество нулевых байтов в области mas (листинг 10.2).

Листинг 10.2. Подсчет числа нулевых элементов

```
<1> ;prg_10_2.asm
<2> model small
<3> .stack 100h
<4> .data
<5> len equ 10            ;количество элементов в mas
<6> mas db 1,0,9,8,0,7,8,0,2,0
<7> .code
```

продолжение ⇨

Листинг 10.2 *(продолжение)*

```
<8>start:
<9>     raov    ax,@data
<10>        mov ds,ax
<11>        mov cx,len       ;длину поля mas в cx
<12>        xor ax.ax
<13>        xor si,si
<14>    cycl:
<15>        jcxz exit        ;проверка cx на 0, если 0, то выход
<16>        cmp mas[si],0    ;сравнить очередной элемент mas с 0
<17>        jne ml           ;если не равно 0, то на ml
<18>        inc al           ;в al - счетчик нулевых элементов
<19>    ml:
<20>        inc si           ;перейти к следующему элементу
<21>        dec cx           ;уменьшить cx на 1
<22>        jmp cycl
<23>    exit:
<24>        mov ax,4c00h
<25>        int 21h          ;возврат управления операционной системе
<26>    end start
```

Цикл в листинге 10.2 организован тремя командами, JCXZ, DEC и JMP (строки 15, 21 и 22). Команда JCXZ выполняет здесь две функции: предотвращает выполнение «пустого» цикла (когда счетчик цикла в CX равен нулю) и отслеживает окончание цикла после обработки всех элементов поля mas. Команда DEC после каждой итерации цикла уменьшает значение счетчика в регистре CX на 1. Заметьте, что при такой организации цикла все операции по его организации выполняются «вручную». Но, учитывая важность такого алгоритмического элемента, как цикл, разработчики процессора ввели в систему команд группу из трех команд, облегчающую программирование циклов. Эти команды также используют регистр ECX/CX как счетчик цикла. Дадим краткую характеристику этим командам.

Команда LOOP позволяет организовать циклы *(loops),* подобные циклам for в языках высокого уровня с автоматическим уменьшением счетчика цикла. Синтаксис команды:

```
loop метка_перехода
```

Команда реализует описанные далее действия.

1. Декремент регистра ECX/CX.
2. Сравнение регистра ECX/CX с нулем:

   П если (ECX/CX) > 0, то управление передается на метку перехода;

   П если (ECX/CX) = 0, то управление передается на следующую после LOOP команду.

Команды LOOPE и LOOPZ (Loop still cx <> 0 or Zero flag = 0 — повторить цикл пока CX <> 0 или ZF = 0) — абсолютные синонимы, поэтому используйте ту команду, которая вам больше нравится. Синтаксис команд:

```
loope/loopz метка_перехода
```

Команды реализуют описанные далее действия.

1. Декремент регистра ECX/CX.
2. Сравнение регистра ECX/CX с нулем и анализ состояния флага нуля ZF:

   П если (ECX/CX) > 0 и ZF = 1, управление передается на метку перехода;

D если (ECX/CX) = 0 или ZF = 0, управление передается на следующую после ШОР команду.

Команды LOOPNE и LOOPNZ (Loop still cx <> 0 or NonZero flag = 0 — повторить цикл, пока CX <> 0 или ZF = 1) также абсолютные синонимы. Синтаксис команд:

```
loopne/loopnz метка_перехода
```

Команды реализуют описанные далее действия.

1. Декремент регистра ECX/CX.
2. Сравнение регистра ECX/CX с нулем и анализ состояния флага нуля ZF:
   - если (ECX/CX) > 0 и ZF = 0, управление передается на метку перехода;
   - если (ECX/CX) = 0 или ZF = 1, управление передается на следующую после LOOP команду.

Команды LOOPE/LOOPZ и LOOPNE/LOOPNZ по принципу своей работы являются **взаимнообратными.** Они расширяют действие команды LOOP тем, что дополнительно анализируют флаг ZF. Это дает возможность организовать досрочный выход из цикла, используя этот флаг в качестве индикатора. Типичное применение этих команд связано с операцией поиска определенного значения в последовательности или со сравнением двух чисел.

Недостаток команд организации цикла LOOP, LOOPE/LOOPZ и LOOPNE/LOOPNZ заключается в том, что они реализуют только короткие переходы (от -128 до +127 байт). Для работы с длинными циклами придется использовать команды условного перехода и команду JMP (см. листинг 10.2), поэтому постарайтесь освоить оба способа организации циклов. Рассмотрим несколько примеров организации циклов с помощью команд LOOP, LOOPE/LOOPZ и LOOPNE/LOOPNZ.

Программа из листинга 10.2 с использованием команды организации цикла будет выглядеть так, как показано в листинге 10.3.

Листинг 10.3. Подсчет нулевых байтов с использованием команд управления циклом

```
;prg_10_3.asm
model small
.stack  100h
.data
len equ 10                  ;количество элементов в mas
mas db 1,0,9,8,0,7,8,0,2,0
.code
start:
    mov ax,@data
        mov ds,ax
        mov cx,len          ;длину поля mas в cx
        xor ax,ax
        xor si,si
        jcxz exit           ;проверка cx на 0, если 0, то выход
    cycl:
        cmp mas[si],0       ;сравнить очередной элемент mas с 0
        jne ml              ;если не равно 0, то на ml
        inc al              ;в al - счетчик нулевых элементов
    ml:
        inc si              ;перейти к следующему элементу
        loop cycl
    exit:
```

продолжение

Листинг 10.3 *(продолжение)*

```
<23>        mov ax,4c00h
<24>        int 21h         ;возврат управления операционной системе
<25>    end start
```

Заметьте, что у команды JCXZ в строке 14 осталась только одна функция — не допустить выполнения «пустого» цикла, поэтому несколько изменилось ее место в тексте программы: теперь она стоит перед меткой начала цикла cycl. Изменение и контроль содержимого регистра CX в процессе выполнения каждой итерации выполняет команда ШОР (строка 21).

Рассмотрим пример, в котором продемонстрируем типичный подход к использованию команды LOOPNZ. В программе из листинга 10.4 ищется первый нулевой элемент в поле mas. Интерес представляют строки 20 и 21. Команда LOOPNZ на основании содержимого регистра CX и флага ZF принимает решение о продолжении цикла. Выход из цикла происходит в одном из двух случаев: CX = 0 (просмотрены все элементы поля mas) или ZF = 1 (командой CMP обнаружен нулевой элемент). Назначение следующей команды JZ (строка 21) в том, чтобы распознать конкретную причину выхода из цикла. Если выход из цикла произошел после просмотра строки, в которой нет нулевых элементов, то флаг JZ не сработает и будет выдано сообщение об отсутствии нулевых элементов в строке (строки 7, 23-25). Если выход из цикла произошел в результате обнаружения нулевого элемента, то в регистре SI окажется номер позиции этого элемента в поле mas и при необходимости можно продолжить обработку. В нашем случае мы просто завершаем программу — переходим на метку exit.

Листинг 10.4. Пример использования команды loopnz

```
<1> ;prg_10_4.asm
<2> model small
<3> .stack  100h
<4> .data
<5> len equ 10   ;количество элементов в mas
<6> mas db  1,0,9,8,0,7,8,0,2,0
<7> message db  "В поле mas нет элементов, равных нулю,$"
<8> .code
<9> start:
<10>        mov ax,@data
<11>        mov ds,ax
<12>        mov cx,len      ;длину поля mas в cx
<13>        xor ax,ax
<14>        xor si,si
<15>        jcxz exit       ;проверка cx на 0, если 0, то выход
<16>        mov si,-1       ;готовим si к адресации элементов поля mas
<17>    cycl:
<18>        inc si
<19>        cmp mas[si],0   ;сравнить очередной элемент mas с 0
<20>        loopnz  cycl
<21>        jz  exit;выяснение причины выхода из цикла
<22>                        ;вывод сообщения, если нет нулевых элементов в mas
<23>        mov ah,9
<24>        mov dx,offset message
<25>        int 21h
<26>    exit:
<27>        mov ax,4c00h
<28>        int 21h         ;возврат управления операционной системе
<29>    end start
```

Читатели, имеющие даже небольшой опыт программирования на языках высокого уровня, знают, что очень часто возникает необходимость во вложенных циклах. Самый простой пример — обработка двухмерного массива. Работу с массивами, в том числе двухмерными, мы рассмотрим в главе 13, пока же разберемся с основными принципами организации вложенных циклов. Основная проблема, которая при этом возникает, — как сохранить значения счетчиков в регистре ECX/CX для каждого из циклов. Для временного сохранения счетчика внешнего цикла на время выполнения внутреннего доступно несколько способов: задействовать регистры, ячейки памяти или стек. В следующем фрагменте программы имеется три цикла, вложенные один в другой. Этот фрагмент можно рассматривать как шаблон для построения других программ с вложенными циклами.

```
...
    mov cx,100        количество повторений цикла cycl_1
 cycl_1:
    push cx           ;счетчик цикла cycl_1 в стек
    ;...команды цикла cycl_1
    mov cx,50         количество повторений цикла cycl_2
cycl_2:
    push cx           ;счетчик цикла cycl_2 в стек
    ;...;команды цикла cycl_2
        mov cx,25     количество повторений цикла cycl_3
    cycl_3:
    ;...;команды цикла cycl_3
        loop cycl_3
    ;...;команды цикла cycl_2
        pop cx        ;восстановить счетчик цикла cycl_2
        loop cycl_2
    ;...;команды цикла cycl_1
        pop cx        ;восстановить счетчик цикла cycl_1
        loop cycl_1
    ;...
```

В качестве примера рассмотрим фрагмент программы, которая обрабатывает специальным образом некоторую область памяти (листинг 10.5). Область памяти рассматривается как совокупность пяти полей, содержащих 10 однобайтовых элементов. Требуется заменить все нулевые байты в этой области значением 0ffh.

Листинг 10.5. Пример использования вложенных циклов

```
;prg_10_5.asm
model small
.stack 100h
.data
mas db 1,0,9,8,0,7,8,0,2,0
    db 1,0,9,8,0,7,8,0,2,0
    db 1,0,9,8,0 7 8,0.?,0
    db 1 0,9,8,0 7 8,0.?,0
    db 1,0,9,8,0,7,8,0,2,0
    .code
    start:
        mov ax,@data
        mov ds,ax
        xor ax,ax
        lea bx,mas
        mov cx,5
    cycl_1:
        push cx
        xor si,si
```

продолжение ⇨

Листинг 10.5 *(продолжение)*

```
        mov cx,10
    cycl_2:
        cmp byte ptr [bx+si],0
        jne no_zero
        mov byte ptr [bx+si],0ffh
    no_zero:
        inc si
        loop cycl_2
        pop cx
        add bx,10
        loop cycl_1
    exit:
        mov ax,4c00h
        int 21h     ;возврат управления операционной системе
    end start
```

Ограничение области видимости для меток

Применение команд, описанных в данной главе, предполагает довольно интенсивное использование меток. На определенном этапе работы многие программисты типовые фрагменты кода берут из ранее разработанных программ, реализуя своеобразное фрагментарно-модульное программирование. При этом может оказаться, что фрагменты кода имеют одинаковые метки. Что делать — пересматривать весь текст? Если не проанализировать код с должным вниманием, потом можно провести много часов за работой с отладчиком или, что еще хуже, программа начнет неправильно работать у заказчика. Поэтому полезными могут оказаться некоторые средства работы с метками, описанные в этом разделе.

Пакет TASM поддерживает директиву LOCALS, разрешающую использовать в программе *локальные блоковые метки*. Формат директивы:

```
LOCALS [символ_префикса]
```

Операнд **символ_префикса** определяет двухсимвольное имя, которое впоследствии потребуется при автоматическом формировании меток в качестве их первых символов. По умолчанию в качестве символа префикса используется двухсимвольная комбинация @@.

Благодаря механизму *локальных блоковых меток* можно в пределах одной программы, но в разных блоках, использовать одинаковые метки. В качестве блока здесь понимаются две конструкции — процедуры и фрагменты программы между двумя обычными метками. Структурно вариант с процедурами выглядит так:

```
model   small
.data
...
LOCALS          ;@@ - префикс локальных меток по умолчанию
prod   proc
@@m1:
...
prod   endp
proc2   proc
@@m1:
...
proc2   endp
.code
...
```

Область видимости локальных блоковых меток также можно ограничить обычными метками. Для примера возьмем фрагмент последней программы и поставим в ней бессмысленные с точки зрения логики, но наглядные в контексте нашего обсуждения локальные блоковые метки:

```
        LOCALS
        ;...
<16>        mov cx,5
<17>    cycl_1:
<18>    @@m1:   jmp @@ml
        ;...
<21>    cycl_2:
        @@m1:   jmp @@m1
        ;...
<25>    no_zero:
        @@m1:   jmp @@ml
        ;...
```

Результат трансляции будет положительным, одноименные локальные блоковые метки @@ml будут интерпретированы транслятором как разные.

Пакет MASM также поддерживает механизм определения локальных блоковых меток, позволяя определить три ближние метки с предопределенными именами: @@, @F и @B. Между этими метками существует связь. Определение метки с именем @@ указывает транслятору на необходимость заменить ее уникальной меткой в форме @@xxxx, где xxxx — уникальное в пределах текущей программы шестнадцатеричное значение. После того как такая метка определена, на нее можно ссылаться с помощью неизменяемых меток @F и @B:

```
;...
@@:
;...
jmp@F    ;ссылка на предыдущую метку @@ (наверх)
jmp@B    ;ссылка на следующую метку @@ (вниз)
;...
@@:
```

Итоги

- Язык ассемблера (система команд процессора) имеет довольно полный и гибкий набор средств организации всевозможных переходов как в пределах текущего сегмента кода, так и во внешние сегменты.
- При организации безусловных переходов возможны переходы как с потерей (JMP), так и с запоминанием (CALL) информации о точке передачи управления.
- Принцип работы команд условного перехода основан на том, что процессор по результатам выполнения некоторых команд устанавливает определенные флаги в регистре EFLAGS/FLAGS. Команды условного перехода анализируют состояние этих флагов и инициируют передачу управления, исходя из результатов анализа.
- Система команд процессора допускает программирование циклов со счетчиком. Для этого используется регистр ECX/CX, в который до входа в цикл должно быть загружено значение счетчика цикла.
- Удобство работы с метками в программе можно повысить, если использовать локальные блоковые метки. Такую возможность предоставляют оба пакета ассемблера: TASM и MASM.

Глава 11

Программирование типовых управляющих структур

- Программирование операторов условного перехода if-else
- Программирование операторов выбора switch
- Программирование операторов цикла while, do-while и for

Язык ассемблера — язык машинных команд. Он поддерживает лишь базовые механизмы организации программ. В нем отсутствуют управляющие конструкции, естественные для языков высокого уровня. Речь идет о поддержке конструкций типа операторов выбора, организации циклов и т. п. В прошлой главе мы положили начало обсуждению этих вопросов, рассмотрев принципы организации циклов в программах на ассемблере. Цель данной главы — разработать шаблоны управляющих конструкций на ассемблере, аналогичные типовым операторам языка высокого уровня.

Поступим просто — откроем любой учебник по языку С или C++, составим список приведенных в нем управляющих операторов и покажем способы их реализации на ассемблере. Типовой список будет следующим:

- операторы выбора:
  - условный оператор if-else;
  - переключатель switch;
- операторы цикла:
  - цикл с предусловием while;
  - цикл с постусловием do-while;
  - итерационный цикл for;
- операторы continue и break.

Условный оператор if-else

Условный оператор if-else используется для принятия решения о дальнейшем пути исполнения программы. Синтаксис условного оператора в нотации языков С и С++:

```
if (выражение)
    оператор_1;
else
    оператор_2
```

Алгоритм работы условного оператора — вычисляется логическое значение выражения: если оно истинно, то выполняется оператор_1, в противном случае — оператор_2.

В общем случае условный оператор может состоять из одного блока if (без блока else):

```
if (выражение)
    оператор_1;
```

В программе на ассемблере данные варианты условного оператора можно реализовать следующим образом:

```
;короткий вариант оператора if (выражение) оператор_1;
    cmp opl,op2 ;вычисление выражения
    jne endif
;...              ;последовательность команд, соответствующая оператор_1
endif:            ;конец короткого условного оператора
;...
```

Строго говоря, использование команды CMP при реализации условного оператора не является обязательным. В данном случае она скорее обозначает место вычисления некоторого условия в программе, по результатам которого принимается решение о ветвлении. Вместо данной команды можно использовать любую команду, изменяющую флаг, который будет анализироваться последующим оператором условного перехода. Эти же рассуждения касаются и команды JNE, вместо которой может стоять требуемая в данном вычислительном контексте команда условного перехода:

```
;полный вариант оператора if (выражение) оператор_1; else оператор_2
    cmp opl,op2 ;вычисление выражения
    jne elsel
;...              ;последовательность команд, соответствующая оператор_1
    jmp endif
elsel:
;...              ;последовательность команд, соответствующая оператор_2
endif:            ;конец полного условного оператора
;...
```

Остается лишь добавить, что приведенные ранее рассуждения о командах CMP и JNE соответствуют также полной форме оператора if.

Оператор выбора switch

Оператор switch (переключатель) в программе на ассемблере можно реализовать несколькими способами. В данной главе мы рассмотрим два наиболее общих и распространенных способа — с использованием команд сравнения и с использованием таблицы. Тем не менее на практике можно встретить и другие способы реализации оператора switch, наиболее полно отражающие условия конкретной решаемой задачи (см., например [8] и некоторые примеры в данной книге).

В нотации языков С и С++ синтаксис оператора switch выглядит так:

```
switch (константное_значение_"выражение") {
   case константное выражение_1:   операнд_1
   case константное выражение_2:   оп.еранд_2
default:    операнд_def ault
>
```

Простейший способ реализации оператора switch в программе на ассемблере заключается в организации последовательных сравнений с условными переходами. К примеру, необходимо обработать три возможные альтернативы в программе, которые идентифицируются целочисленными значениями 1, 2, 3("выражение"):

```
;...     ;формирование константное_значение_"выражение"
mov al,константное_значение_"выражение"
cmp al,l          ;проверка первой альтернативы (константное выражение_1=1?)
]e handle_conditionl   ;равно, идем на обработку
cmp al,2          ;проверка второй альтернативы (константное выражение_2=2?)
je handle_condition2   ;равно, идем на обработку
cmp al,3          ;проверка третьей альтернативы (константное выражение_3=3?)
je   handle_condition2    ;равно, идем на обработку
;если condition<>1|2|3, то производим обработку по умолчанию
;...             ;последовательность команд для обработки по умолчанию
   jmp end_switch      ;уходим на конец switch
handle_conditionl:
;последовательность команд для обработки handle_conditionl
;...
   jmp end_switch      ;уходим на конец switch
handle_condition2:
;последовательность команд для обработки handle_condition2
;...
   jmp end_switch      ;уходим на конец switch
handle_condition3:
;последовательность команд для обработки handle_condition3
;...
end_switch:
;...             ;продолжение программы
```

Второй, более элегантный способ реализации конструкции, соответствующей оператору switch, — табличный. Сразу отмечу, что данный материал логичнее рассматривать после изучения главы 13, посвященной сложным структурам данных. Поэтому, если у вас возникнут трудности с пониманием представленного здесь материала, пропустите его и вернитесь к нему позже.

Для реализации рассматриваемого способа в памяти моделируется некоторое подобие таблицы. Каждая строка таблицы состоит из двух ячеек. Строки таблицы располагаются в памяти последовательно — друг за другом. В первой ячейке таблицы располагается значение константное_выражение_n, во второй ячейке — адрес перехода, если выполняется условие константное_значение_"выражение" = константное_выражение_n. Целью адреса перехода может быть процедура или метка, соответствующая фрагменту кода, обрабатывающему выполнение условия, заданного в заголовке оператора выбора. Реализация табличным способом приведенного ранее примера будет выглядеть так:

```
.data
table_switch    db  1
   dw  handle_conditionl
   db  2
   dw  handle_condition2
   db  3
   dw  handle_condition3
```

```
;...
.code
;...    ;формирование константное_значение_"выражение"
    mov al,константное_значение_"выражение"
    mov bx,offset   table_switch
    mov cx,3                ;счетчик циклов=количество альтернатив
next_condition:
    cmp al,[bx]            ;проверка очередной альтернативы
    jne next_              ;к следующему элементу
    jmp word ptr  [bx+1]    ;на обработку совпадения
    add bx,3               ;адрес следующей строки таблицы -> bx
next_:  loopnext_condition
;default:
;если condition<>1|2|3, то производим обработку по умолчанию
;...    ;последовательность команд для обработки по умолчанию
    jmp ...                ;уходим, куда нужно
handle_condition1:
;последовательность команд для обработки handle_condition1
;...
    jmp ...                ;уходим, куда нужно
handle_condition2:
;последовательность команд для обработки handle_condition2
;...
    jmp ...                ;уходим, куда нужно
handle_condition3:
;последовательность команд для обработки handle_condition3
;...
    jmp ...                ;уходим, куда нужно
;...                        ;продолжение программы
```

Для лучшего структурирования программы последовательность команд для обработки условия handle_conditionN можно оформить в виде процедур:

```
.data
table_switch    db  1
    dw  handle_condition1    ;адрес процедуры handle_condition1
    db  2
    dw  handle_condition2    ;адрес процедуры handle_condition2
    db  3
    dw  handle_condition3    ;адрес процедуры handle_condition3
;...
handle_condition1   proc
;...    ;последовательность команд для обработки handle_condition1
    endp
handle_condition2   proc
;...    ;последовательность команд для обработки handle_condition2
    endp
handle_condition3   proc
;...    ;последовательность команд для обработки handle_condition3
    endp
.code
;...    ;формирование константное_значение_"выражение"
    mov al,константное_значение_"выражение"
    mov bx,offset   table_switch
    mov cx,3                ;счетчик циклов=количество альтернатив
next_condition:
    cmp al,[bx]            ;проверка очередной альтернативы
    jne next_              ;к следующему элементу
    call word ptr  [bx+1]   ;на обработку совпадения
    jmp end_switch
    add bx,3               ;адрес следующей строки таблицы -> bx
next_:  loopnext_condition
;default:
;если condition<>1|2|3, то производим обработку по умолчанию
```

```
;...                ;последовательность команд для обработки по умолчанию
end_switch:
;...                ;продолжение программы
```

Операторы цикла

В предыдущей главе мы рассматривали общую концепцию реализации циклов в программах на языке ассемблера. В данном разделе обсудим способы организации циклов, традиционно поддерживаемых большинством языков высокого уровня. Для полноты изложения нам, возможно, придется повторить некоторые тезисы предыдущей главы.

Оператор цикла с предусловием while

Оператор цикла с предусловием while имеет следующий формат:

```
while (выражение-условие)
    тело цикла
```

Для облегчения реализации этого оператора приведем его псевдокод, близкий к нотации ассемблера:

```
while_begin:
;вычисление логического значения "выражение-условие"
ЕСЛИ "выражение-условие"=FALSE ТО ПЕРЕЙТИ НА end_while
операторы цикла
...
ПЕРЕЙТИ на while_begin  ;переход на начало очередной итерации цикла
end_while:              ;конец цикла
;...                    ;продолжение программы
```

На ассемблере это может выглядеть так:

```
while_begin:
    cmp al,1;вычисление логического значения "выражение-условие"
    jne end_while       ;ЕСЛИ не равно ТО ПЕРЕЙТИ НА end_while
...
;что-то делаем - тело цикла
...
    jmp while_begin     ;переход на начало очередной итерации цикла
end_while:              ;конец цикла
;...                    ;продолжение программы
```

Операторы continue и break

Прекращение выполнения цикла возможно в двух случаях:

- значение выражение-условие (см. выше) равно FALSE;
- имеет место безусловная передача управления за пределы цикла (в программе на языках С и С++ эти действия выполняют операторы break и goto).

В языках С и С++ изменить последовательность выполнения тела цикла можно с помощью оператора continue.

Алгоритм выполнения операторов break, goto и continue прост, они передают управление в определенную точку программы:

- break — сразу за конец цикла;
- goto — в произвольное место программы;
- continue — на начало последнего открытого оператора while, do или for.

В программе на ассемблере естественнее всего реализовать эти операторы с помощью команды безусловного перехода JMP, которая передаст управление в нужную точку программы.

Оператор цикла с постусловием do-while

Продолжим обсуждение операторов цикла различного типа. Очередной — оператор цикла с постусловием do-while. Его формат в нотации языков С и С++:

```
do
    тело цикла
while(выражение-условие)
```

Основная отличительная особенность данного оператора — в том, что операторы, составляющие тело цикла, выполняются хотя бы один раз. Близкий к нотации ассемблера псевдокод реализации этого оператора выглядит так:

```
do_while_begin:
операторы цикла
...
;вычисление логического значения "выражение-условие"
ЕСЛИ "выражение-условие"=FALSE ТО ПЕРЕЙТИ НА end_while
ПЕРЕЙТИ на do_while_begin:   ;переход на начало очередной итерации цикла
end_while:                   ;конец цикла
;...                        ;продолжение программы
```

Оператор итерационного цикла for

Оператор итерационного цикла for — это единственный оператор цикла, хотя бы частично поддерживаемый ассемблером на уровне команд. Его формат в нотации языков С и С++:

```
for (инициализация цикла; выражение-условие; выражение)
    тело цикла
```

С точки зрения грамматики все три выражения в операторе for являются произвольными. Но обычно первое из них (инициализация цикла) предназначено для присвоения начального значения переменной цикла, второе (выражение-условие) вычисляет логическое значение для принятия решения о выходе из цикла, третье (выражение) вычисляет значение переменной цикла на очередной его итерации.

Ассемблер с помощью команды ШОР поддерживает только один вариант организации такого цикла, который был подробно рассмотрен в предыдущей главе. Для реализации произвольного варианта цикла for может оказаться полезным следующий псевдокод:

```
инициализация переменной цикла
for_begin:
;вычисление логического значения "выражение-условие"
ЕСЛИ  "выражение-условие"=FALSE ТО ПЕРЕЙТИ НА end_for
операторы цикла
...
вычисляем выражение - очередное значение переменной цикла
ПЕРЕЙТИ на for_begin:    ;переход на начало очередной итерации цикла
end_for:                 ;конец цикла
;...                    ;продолжение программы
```

Какой цикл эффективнее? Тот, в котором меньше проверок и переходов.

Особенности пакета MASM

Пакет MASM имеет встроенные средства, визуально приближающие написанную программу к программам на языке высокого уровня. Это директивы для формирования конструкций, аналогичных условным и циклическим операторам языков высокого уровня: .IF, .ELSE, .ELSEIF, .ENDIF, .REPEAT, .UNTIL, .WHILE, .ENDW, .BREAK, .CONTINUE. Всем директивам предшествует символ . (точка). Использование данных конструкций можно рекомендовать для повышения надежности кода. Программист, оставаясь на уровне ассемблера, повышая логический уровень контролируемых им синтаксических конструкций, снижает вероятность ошибок логики, ускоряя тем самым весь процесс разработки.

Условная конструкция .IF

Конструкция .IF предназначена для генерации кода, функционально аналогичного условному оператору языков высокого уровня. Ее синтаксис:

```
.IF условие
   предложения блока IF
   [.ELSEIF] условие
   предложения блока ELSEIF
...
   [.ELSE]
   предложения блока ELSE
.ENDIF
```

Здесь слова .IF, .ELSEIF, .ELSE, .ENDIF — служебные, аусловие представляет один из следующих операторов:

- оп1 == оп2 — операнды равны;
- оп1 != оп2 — операнды не равны;
- оп1 > оп2 — больше;
- оп1 >= оп2 — больше или равно;
- оп1 < оп2 — меньше;
- оп1 <= оп2 — меньше или равно;
- оп1 & номер_бита — проверка бита;
- ! onl — инверсия (NOT);
- оп1 && оп2 — логическое умножение (AND);
- onl || on2 — логическое сложение (OR);
- CARRY? — EFLAGS.CF=1?;
- OVERFLOW? - EFLAGS.OF=1?;
- PARITY? - EFLAGS.PF=1?;
- SIGN? - EFLAGS.SF=1?;
- ZERO? - EFLAGS.ZF=1?.

Эти же условия используются в конструкциях .REPEAT, .WHILE, .BREAK, .CONTINUE (см. далее). Сочетание операндов оп1 и оп2 должно быть допустимым для команд ассемблера, то есть выражение типа obl_1<obl_2 (память-память) ассемблер не пропустит.

Допускается вложение конструкций .IF. Назначение и использование директив .ELSE и .ELSEIF аналогичны соответствующим конструкциям в языках высокого уровня. Завершающая директива условного блока .ENDIF обязательна.

Интерес для изучения представляет то, как ассемблер преобразует директивы рассматриваемых нами конструкций высокого уровня в машинный код. Для этого необходимо вставить в любую программу, не имеющую синтаксических ошибок, фрагмент, подобный следующему:

```
...
.if al==0
mov      ax,1
.elseif ax==0
mov      ax,2
.else
mov      ax,3
.endif
...
```

После этого необходимо выполнить компиляцию примера:

```
ML.EXE  /Fl /Zi Prg_6_1.asm
```

Ключ /Fl предназначен для генерации файла листинга (по умолчанию не создается). Он намеренно вставлен в командную строку для того, чтобы *не* увидеть замену высокоуровневых директив командами ассемблера в тексте листинга. Для визуализации замены встроенных макросов необходимо загрузить исполняемый модуль в отладчик CodeView, выбрать в окне Source команду Options ▶ Source Window и в открывшемся окне Source Window Options выбрать режим отображения (Display Mode) Mixed Source and Assembly. Взору предстанет следующий смешанный код:

```
25:  .if al==0
1792:000C 36803E000100  CMP BYTE PTR SS:[0100],00
1792:0012 7505  JNZ 0019
26: mov ax,1
1792:0014 B80100    MOV AX,0001
27:  .elseif ax==0
1792:0017 EB0C  JMP 0025
1792:0019 0BC0  OR  AX,AX
1792:001B 7505  JNZ 0022
28: mov ax,2
1792:001D B80200    MOV AX,0002
29:  .else
1792:0020 EB03  JMP 0025
30: mov ax,3
1792:0022 B80300    MOV AX,0003
31:  .endif
```

На примере команд сравнения видно, как ассемблер выполняет оптимизацию кода. Для приведенного фрагмента программы ассемблер реализовал сравнение в двух вариантах:

- для переменной al==0 - CMP BYTE PTR SS:[0100],00;
- для регистра ax==0 — OR AX,AX.

Циклическая конструкция .REPEAT

Конструкция .REPEAT соответствует циклу с постусловием в языках высокого уровня.

Синтаксис конструкции:

```
.REPEAT
команды ассемблера
.UNTIL условие
```

Существует и второй тип конструкции .REPEAT:

```
.REPEAT
команды ассемблера
.UNTILCXZ [условие]
```

Конструкция .REPEAT заставляет процессор повторно исполнять команды ассемблера пока условие истинно (не равно нулю). Особенностью такого цикла является то, что блок команд ассемблера выполняется хотя бы один раз. Исходя из конкретного условия, ассемблер при формировании его машинной реализации выполняет необходимую оптимизацию. Что именно при этом происходит, можно увидеть, загрузив код в CodeView указанным ранее способом.

Директива .UNTILCXZ может использоваться с проверкой дополнительного условия в виде выражение1 == выражение2 или выражение1 != выражение2, чтобы генерировать команды ШОРЕ и LOOPNE. Директива .UNTILCXZ выполняет декремент регистра ЕСХ и прекращает выполнение цикла при ЕСХ=0. При отсутствии условия директива .UNTILCXZ просто генерирует команду LOOP.

Циклическая конструкция .WHILE

Конструкция .WHILE соответствует циклу с предусловием в языках высокого уровня.

Синтаксис конструкции:

```
.WHILE условие
команды ассемблера
.ENDW
```

Директива .WHILE генерирует последовательность команд выполняющих блок команд ассемблера, если условие истинно. Директива .ENDW завершает блок команд ассемблера. Он заставляет вернуть управление конструкции .WHILE, которая после проверки истинности условия либо заново инициирует выполнение блока команд ассемблера, либо передает управление команде, следующей за .ENDW. Конструкция .WHILE допускает вложенность, причем ассемблер выполняет необходимую оптимизацию для формирования наиболее эффективного кода.

Конструкции .BREAK и .CONTINUE

Конструкции .BREAK и .CONTINUE соответствуют аналогичным конструкциям языков высокого уровня.

Директива .BREAK генерирует код выхода на первую команду, следующую за телом текущего цикла .WHILE или .REPEAT. Ее синтаксис:

```
.BREAK [.IF условие]
```

Директива .BREAK позволяет задать дополнительное условие, при выполнении которого будет выполнен выход из цикла независимо от условия основного цикла.

Директива .CONTINUE передает управление на начало текущего цикла .WHILE или .REPEAT. Ее синтаксис:

```
.CONTINUE [.IF условие]
```

При необходимости директива .CONTINUE может определять дополнительное условие, при соблюдении которого будет выполнен переход на начало текущего цикла.

Директивы .CONTINUE и .BREAK могут использоваться только внутри циклов .WHILE или .REPEAT.

Комплексный пример

Чтобы проиллюстрировать использование директив MASM для имитации конструкций высокого уровня, рассмотрим пример. Этот пример не является примером «вещи в себе», он призван показать класс задач, которые актуальны при программировании на любом языке. Это задачи контроля информации, поступающей в программу в ходе ее функционирования. Как методически правильно подойти к этому вопросу, не «изобретая каждый раз заново велосипед»? Если вы знакомы с теорией построения трансляторов, то суть проблемы разбора символьных последовательностей и подходы к ее решению для вас не новы. Если же у вас таких знаний нет, то материал данного раздела будет полезен для формирования у вас теоретически грамотного подхода к решению проблемы контроля входных данных ваших программ.

Частью любого компилятора является *сканер,* задача которого — чтение входного потока символов и выделение в нем синтаксически значимых единиц информации. Примером таких единиц могут быть ключевые слова языка, имена переменных, константы и другие синтаксически автономные единицы исходной программы. Сканер распознает их и в зависимости от их типа формирует внутреннее представление этих единиц и информацию о них для дальнейшей синтаксической и семантической обработки. Теоретической основой работы сканера является теория автоматов.

Немного теории. Конечный автомат представляет собой кортеж M = {K, A, P, S, F}, где:

- K — конечное множество состояний;
- A — конечный входной алфавит;
- P — множество переходов;
- S — начальное состояние;
- F — множество конечных (последних) состояний.

Принципиально сканер функционирует так. Он имеет конечное множество состояний K, одно из которых является начальным S, и несколько конечных F. Перед началом обработки символов очередной лексемы сканер находится в состоянии S. По мере считывания очередной литеры лексемы состояние сканера меняется. Эти переходы также заранее определены множеством правил перехода P. После окончания чтения лексемы автомат должен оказаться в одном из конечных состояний, некоторые из которых могут соответствовать состоянию ошибки. Исходя из того, в каком из состояний оказался сканер, он делает вывод о принадлежности лексемы входному языку. Если лексема ошибочная, то компилятором формируется соответствующее диагностическое сообщение, если же лексема корректная, то она в зависимости от своего типа подвергается дальнейшей обработке.

Рассмотрим пример конечного автомата для распознавания целых десятичных чисел со знаком, считываемых программой с консоли. Хорошую помощь в построении конечных автоматов оказывают формы Бэкуса–Наура и синтаксические

диаграммы. Их мы обсуждали в главе 5. Формы Бэкуса–Наура для построения десятичных чисел со знаком выглядят так:

```
<десятичное_знаковое_целое>⇒<число_без_знака>|+<число_без_знака>|<число_без_знака>
<число_без_знака>⇒<дес_цифра>|<число_без_знака><дес_цифра>
<дес_цифра>⇒0|1|2|3|4|5|6|7|8|9
```

Графически конечный автомат удобно представлять в виде направленного графа. Конечный автомат для десятичного числа со знаком показан на рис. 11.1.

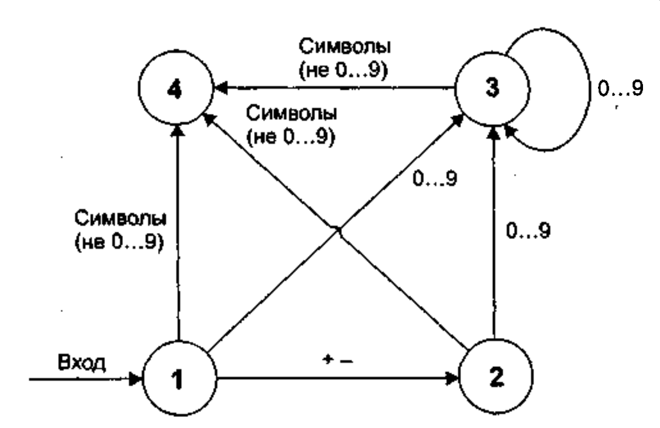


Рис. 11.1. Конечный автомат для десятичного числа со знаком

По рисунку легко закончить формальное определение конечного автомата для десятичного числа со знаком:

- K = {1, 2, 3, 4};
- A -{+,-, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
- P = {1->2, 1->3, 1->4, 2->4, 2->3, 3->4, 3->3};
- S = {1};
- F = {4}

Далее приведена программа (листинг 11.1), моделирующая конечный автомат для распознавания десятичных чисел со знаком.

Листинг 11.1. Программа конечного автомата для десятичных чисел со знаком

```
.model  small          ; модель памяти
.data                  ; сегмент данных
message db  0dh,0ah,'I am ready to receive your variant of signed decimal
number (to finish press Enter): $'
message_Invalid_Number  db  0dh,0ah,'Your input is bed, please input your
number again or press Enter. $'
good_luck_message   db  0dh,0ah,'Your number is: '
input_string    db  10 dup (20h) ;строка для ввода 9 цифр и знака
    db  0dh,0ah,'Thank you for use this program. I wish you good luck...$'
.stack  256h           ; сегмент стека
.code                  ; сегмент кода
GetNextChar proc
    mov ah,1           ;чтение символа из стандартного ввода
    int 21h            ;символ в al
    ret
GetNextChar endp
```

```
main    proc                ; начало процедуры main
    mov ax,@data            ; заносим адрес сегмента данных в регистр ах
    mov ds,ax               ; ах в ds
    mov ah,9
    lea dx,message
    int 21h                 ; вывод сообщения на экран
    xor si ,si              ;подготовка индексного регистра si
    call    GetNextChar     ;ввод первого символа в al
.if (al=="+" || al=="-"  || al>=30h && al<=39h)
    mov input_string[si],al
    inc si
.else
    jmp bed_end
.endif
.while si<9
    call    GetNextChar  ;ввод очередного символа в al
.if (al>=30h && al<=39h)
    mov input_string[si],al
    inc si
.continue
.elseif al==0dh
.break
.else
    jmp bed_end
.endif
.endw
    mov ah,9
    lea dx,good_luck_message
    int 21h                 ; вывод сообщения good luck_message на экран
    jmp exit_prog
bed_end:
    mov ah,9
    lea dx,message_Invalid_Number
    int 21h                 ; вывод сообщения message_Invalid_Number на экран
exit_prog:
    call   GetNextChar ;задержка до ввода любого символа
    mov ax,4c00h            ; пересылка 4c00h в регистр ах
    int 21h                 ; вызов прерывания с номером 21h
main    endp                ; конец процедуры main
end main                    ; конец программы с точкой входа main
```

Напомню, что для получения пригодного для отладки исполняемого модуля данной программы необходимо сформировать командную строку вида

```
ML.EXE  /Fl /Zi Prg_11_1.asm  >p
```

Для отладки используйте отладчик CodeView:

```
cv.exe Prg_11_1.asm
```

В качестве упражнения вы можете потренироваться в программировании конечных автоматов для разбора произвольных символьных строк. После освоения материала главы 17 практически полезной может быть программа конечного автомата для разбора чисел с плавающей точкой. Дополнительный материал по программированию конечных автоматов вы найдете в [8].

В заключение данной главы хотелось бы заметить, что приведенные в ней рассуждения о моделировании вариантов типовых циклических конструкций не являются догмой. Их основное назначение — показать возможность и практическую пользу моделирования циклических конструкций в ассемблерной программе. Важность материала данной главы состоит в том, что использование этих конструкций в программах способно, наряду с облегчением написания ассемблерного кода, существенно повысить его надежность.

Итоги

- Язык ассемблер не содержит управляющих конструкций, естественных для языков высокого уровня. Это существенно снижает наглядность ассемблерных программ и надежность программирования в целом.
- На практике существует настоятельная необходимость поддержки любым языком операторов выбора, различного типа операторов цикла и некоторых других.
- В разных пакетах ассемблера реализованы свои «фирменные» подходы к решению данной проблемы:
  - TASM имеет два режима работы: MASM и IDEAL. В режиме MASM все управляющие конструкции нужно моделировать. Этот процесс подробно описан в первой части данной главы. В режиме IDEAL есть встроенные средства для организации типовых управляющих конструкций, но из-за прекращения поддержки TASM фирмой Borland работа в режиме IDEAL теряет актуальность;
  - пакет MASM имеет встроенные средства, моделирующие управляющие конструкции языков высокого уровня. Их использование существенно повышает скорость и надежность разработки программ.
- Важно понимать, что встроенные средства, моделирующие управляющие конструкции языков высокого уровня — это лишь надстройка компилятора над некоторым базовым ассемблерным уровнем. В конечном итоге, все высокоуровневые конструкции будут переведены в последовательность команд ассемблера и исполнены процессором.

Глава 12
Цепочечные команды

- **Средства процессора для обработки цепочек элементов в памяти**
- **Операции пересылки и сравнения цепочек**
- **Операции для работы с отдельными элементами цепочек**
- **Операции для работы с портами ввода-вывода**

В данной главе будет рассмотрена чрезвычайно интересная группа команд, понимание принципов работы которых и их грамотное использование способны значительно облегчить жизнь программисту, пишущему программы на языке ассемблера. Речь идет о так называемых *цепочечных командах,* которые иногда называют *командами обработки строк символов.* Однако эти названия не идентичны. Под строкой символов здесь понимается последовательность байтов, а цепочка — это более общее название для случаев, когда размер элемента последовательности превышает байт и составляет слово или двойное слово. Таким образом, цепочечные команды позволяют проводить действия над блоками памяти, представляющими собой последовательности элементов следующего размера:

- 8 битов, то есть байт;
- 16 битов, то есть слово;
- 32 бита, то есть двойное слово.

Содержимое этих блоков для процессора не имеет никакого значения. Это могут быть символы, числа и все что угодно. Главное, чтобы размеры элементов соответствовали одному из перечисленных ранее вариантов и эти элементы находились в соседних ячейках памяти.

Всего в системе команд процессора поддерживаются семь операций-примитивов обработки цепочек. Каждая из них реализуется в процессоре тремя командами,

в свою очередь, каждая из этих команд работает с соответствующим размером элемента — байтом, словом или двойным словом. Особенность всех цепочечных команд в том, что они, кроме обработки текущего элемента цепочки, корректируют содержимое определенных регистров с тем, чтобы автоматически продвинуться к следующему элементу цепочки.

Перечислим операции-примитивы обработки цепочек и реализующие их команды ассемблера.

- Пересылка цепочки:
  - MOVS адрес_приемника,адрес_источника;
  - MOVSB;
  - MOVSW;
  - MOVSD.
- Сравнение цепочек:
  - CMPS адрес_приемника,адрес_источника;
  - CMPSB;
  - CMPSW;
  - CMPSD.
- Сканирование цепочки:
  - SCAS адрес_приемника;
  - SCASB;
  - SCASW;
  - SCASD.
- Загрузка элемента из цепочки:
  - LODS адрес_источника;
  - LODSB;
  - LODSW;
  - LODSD.
- Сохранение элемента в цепочке:
  - STOS адрес_приемника;
  - STOSB;
  - STOSW;
  - STOSD.
- Получение элементов цепочки из порта ввода-вывода:
  - INS адрес_приемника,номер_порта;
  - INSB;
  - INSW;
  - INSD.
- Вывод элементов цепочки в порт ввода-вывода:
  - OUTS номер_порта,адрес_источника;

- OUTBS;
- OUTWS;
- OUTDS.

Логически к этим командам нужно отнести и так называемые *префиксы повторения.* Вспомните формат машинной команды и его первые необязательные байты префиксов. Один из возможных типов префиксов — это префиксы повторения. Они предназначены для использования цепочечными командами. Префиксы повторения имеют свои мнемонические обозначения:

- REP;
- REPE, или REPZ;
- REPNE, или REPNZ.

Эти префиксы повторения указываются перед нужной цепочечной командой в поле метки. Цепочечная команда без префикса выполняется один раз. Размещение префикса перед цепочечной командой заставляет ее выполняться в цикле. Различия приведенных префиксов — в основании, по которому принимается решение о циклическом выполнении цепочечной команды: по состоянию регистра ECX/CX или по флагу нуля ZF.

- Префикс повторения REP (REPeat) используется с командами, реализующими операции-примитивы пересылки и сохранения элементов цепочек, — соответственно, MOVS и STOS. Префикс REP заставляет данные команды выполняться, пока содержимое в ECX/CX не станет равным 0. При этом цепочечная команда, перед которой стоит префикс, автоматически уменьшает содержимое ECX/CX на единицу. Та же команда, но без префикса, этого не делает.
- Префиксы повторения REPE (REPeat while Equal) и REPZ (REPeat while Zero) являются абсолютными синонимами. Они заставляют цепочечную команду выполняться до тех пор, пока содержимое ECX/cx не равно 0 или флаг ZF равен 1. Как только одно из этих условий нарушается, управление передается следующей команде программы. Благодаря возможности анализа флага ZF наиболее эффективно эти префиксы можно использовать с командами CMPS и SCAS для поиска различающихся элементов цепочек.
- Префиксы повторения REPNE (REPeat while Not Equal) и REPNZ (REPeat while Not Zero) также являются абсолютными синонимами. Их действие на цепочечную команду несколько отличается от действий префиксов REPE/REPZ. Префиксы REPNE/REPNZ заставляют цепочечную команду циклически выполняться до тех пор, пока содержимое ECX/CX не равно нулю или флаг ZF равен нулю. При нарушении одного из этих условий работа команды прекращается. Данные префиксы также можно использовать с командами CMPS и SCAS, но для поиска совпадающих элементов цепочек.

Следующий важный момент, связанный с цепочечными командами, заключается в особенностях формирования физического адреса операндов адрес_источника и адрес_приемника. *Цепочка-источник,* адресуемая операндом адрес_источника, может находиться в текущем сегменте данных, определяемом регистром DS. *Цепочка-приемник,* адресуемая операндом адрес_приемника, должна быть в дополнительном сегменте данных, адресуемом сегментным регистром ES. Важно отметить,

что допускается замена (с помощью префикса замены сегмента) только регистра DS, регистр ES подменять нельзя. Вторые части адресов (смещения цепочек) также находятся в строго определенных местах. Для цепочки-источника это регистр ESI/SI (Source Index register — индексный регистр источника). Для цепочки-получателя это регистр EDI/DI (Destination Index register — индексный регистр приемника). Таким образом, полные физические адреса для операндов цепочечных команд следующие:

- адрес_источника — пара ds:esi/si;
- адрес_приемника — пара es:edi/di.

Кстати, вспомните команды LDS и LES, которые мы рассматривали в главе 7. Эти команды позволяют получить полный указатель (сегмент плюс смещение) на ячейку памяти. Применение их в данном случае очень удобно в силу жесткой регламентации использования регистров для адресации операндов источника и приемника в цепочечных командах.

Вы, наверное, обратили внимание на то, что все семь групп команд, реализующих цепочечные операции-примитивы, имеют похожий по структуре набор команд. В каждом из этих наборов присутствуют одна команда с явным указанием операндов и три команды, не имеющие операндов. На самом деле набор команд процессора имеет соответствующие машинные команды только для цепочечных команд ассемблера без операндов. Команды с операндами транслятор ассемблера задействует только для определения типов операндов. После того как выяснен тип элементов цепочек по их описанию в памяти, генерируется одна из трех машинных команд для каждой из цепочечных операций. По этой причине все регистры, содержащие адреса цепочек, должны быть инициализированы заранее, в том числе и для команд, допускающих явное указание операндов. В силу того, что цепочки адресуются однозначно, нет особого смысла применять команды с операндами. Главное, что вы должны запомнить, — правильная загрузка регистров указателями обязательно требуется до выдачи любой цепочечной команды. Также на практике могут встретиться случаи, когда транслятор потребует явно переопределить сегмент ES: для соответствующего операнда в команде с явным указанием операндов.

Последний важный момент, касающийся всех цепочечных команд, — это направление обработки цепочки. Есть две возможности:

- от начала цепочки к ее концу, то есть в направлении возрастания адресов;
- от конца цепочки к началу, то есть в направлении убывания адресов.

Как мы увидим позже, цепочечные команды сами выполняют модификацию регистров, адресующих операнды, обеспечивая тем самым автоматическое продвижение по цепочке. Количество байтов, на которые эта модификация осуществляется, определяется кодом команды. А вот знак этой модификации определяется значением *флага направления* DF (Direction Flag) в регистре EFLAGS/FLAGS:

- если DF = 0, то значения индексных регистров ESI/SI и EDI/DI будут автоматически увеличиваться (операция инкремента) цепочечными командами, то есть обработка будет осуществляться в направлении возрастания адресов;

- если DF = 1, то значения индексных регистров ESI/SI и EDI/DI будут автоматически уменьшаться (операция декремента) цепочечными командами, то есть обработка будет идти в направлении убывания адресов.

Состоянием флага DF можно управлять с помощью двух команд, не имеющих операндов:

- CLD (Clear Direction Flag) — очистить флаг направления (команда сбрасывает флаг направления DF в 0);
- STD (Set Direction Flag) — установить флаг направления (команда устанавливает флаг направления DF в 1).

Это вся информация, касающаяся общих свойств цепочечных команд. Далее мы более подробно рассмотрим каждую операцию-примитив и команды, которые ее реализуют. При этом одну из команд в каждой группе цепочечных команд (команду с операндами) мы будем рассматривать подробнее, поскольку это более общая команда в смысле ограничений, накладываемых на типы операндов.

Пересылка цепочек

Команды, реализующие операцию-примитив пересылки цепочек, производят копирование элементов из одной области памяти (цепочки) в другую. Размер элемента определяется применяемой командой. Ассемблер предоставляет в распоряжение программиста четыре команды, работающие с разными размерами элементов цепочки:

- MOVS адрес_приемника,адрес_источника — переслать цепочку (MOVe String);
- MOVSB — переслать цепочку байтов (MOVe String Byte);
- MOVSW — переслать цепочку слов (MOVe String Word);
- MOVSD — переслать цепочку двойных слов (MOVe String Double word).

Команда MOVS

Синтаксис команды MOVS:

```
movs адрес_приемника,адрес_источника
```

Команда копирует байт, слово или двойное слово из цепочки, адресуемой операндом адрес_источника, в цепочку, адресуемую операндом адрес_приемника. Размер пересылаемых элементов ассемблер определяет, исходя из атрибутов идентификаторов, указывающих на области памяти приемника и источника. К примеру, если эти идентификаторы были определены директивой DB, то пересылаться будут байты, если идентификаторы были определены с помощью директивы DD, то пересылке подлежат 32-разрядные элементы, то есть двойные слова. Ранее уже было отмечено, что для цепочечных команд с операндами, к которым относится и команда пересылки movs адрес_приемника,адрес_источника, не существует машинного аналога. При трансляции в зависимости от типа операндов транслятор преобразует ее в одну из трех машинных команд: MOVSB, MOVSW или MOVSD.

Сама по себе команда MOVS пересылает только один элемент, исходя из его типа, и модифицирует значения регистров ESI/SI и EDI/DI. Если перед командой написать префикс REP, то одной командой можно переслать до 64 Кбайт данных (если

размер адреса в сегменте 16 бит — use16) или до 4 Гбайт данных (если размер адреса в сегменте 32 бита — use32). Число пересылаемых элементов должно быть загружено в счетчик — регистр CX (use16) или ECX (use32). Перечислим последовательность действий, которые нужно выполнить в программе для того, чтобы переслать цепочку элементов из одной области памяти в другую с помощью команды MOVS. В общем случае эту последовательность можно рассматривать как типовую для выполнения любой цепочечной команды.

1. Установить значение флага DF в зависимости от того, в каком направлении будут обрабатываться элементы цепочки — в направлении возрастания или убывания адресов.
2. Загрузить указатели на адреса цепочек в памяти в пары регистров DS:(E)SI и ES:(E)DI.
3. Загрузить в регистр ECX/CX количество обрабатываемых элементов.
4. Выдать команду MOVS с префиксом REP.

На примере листинга 12.1 рассмотрим, как эти действия реализуются программно. В этой программе производится пересылка символов из одной строки в другую. Строки находятся в одном сегменте памяти. Для пересылки используется команда-примитив MOVS с префиксом повторения REP.

Листинг 12.1. Пересылка строк командой MOVS

```
;prg_12_1.asm
masm
model   small
stack   256
.data
source  db  "Тестируемая строка",'$' ;строка-источник
dest    db  19 dup (" ")             ;строка-приемник
.code
        assume ds:@data,es:@data
main:                                ;точка входа в программу
        mov ax,@data                 ;загрузка сегментных регистров
        mov ds,ax                    ;настройка регистров DS и ES
                                     ;на адрес сегмента данных
        mov es,ax
        cld                          ;сброс флага DF - обработка
                                     ; строки от начала к концу
        lea si,source                ;загрузка в si смещения
                                     ;строки-источника
        lea di,dest                  ;загрузка в DS смещения строки-приемника
        mov cx,20                    ;для префикса rep - счетчик
                                     ;повторений (длина строки)
rep     movs dest,source             ;пересылка строки
        lea dx,dest
        mov ah,09h                   ;вывод на экран строки-приемника
        int 21h
exit:
        mov ax,4c00h
        int 21h
end     main
```

Пересылка байтов, слов и двойных слов

Пересылка байтов, слов и двойных слов производится командами MOVSB, MOVSW и MOVSD. Единственной отличительной особенностью этих команд от команды movs

является то, что последняя может работать с элементами цепочек любого размера — 8, 16 или 32 бита. При трансляции команда MOVS преобразуется в одну из трех команд: MOVSB, MOVSW или MOVSD. Ранее было показано, что решение о том, в какую конкретно команду будет произведено преобразование, принимается транслятором, исходя из размеров элементов цепочек, адреса которых указаны в качестве операндов команды MOVS. Что касается адресов цепочек, то для любой из четырех команд они должны формироваться программой явно и заранее в регистрах ESI/SI и EDI/DI.

К примеру, посмотрим, как изменится программа из листинга 12.1 при использовании команды MOVSB:

```
...
.data
source  db  "Пересылаемая строка$"   ;строка-источник
dest    db  20 DUP (?)               ;строка-приемник
.code
ASSUME  ds              :@data,es:@data
main:
...
    cld                 ;сброс флага DF — просмотр строки от начала к концу
    lea  si,source      ;загрузка в ES строки-источника
    lea  di,dest        ;загрузка в DS строки-приемника
    mov  cx,20          ;для префикса rep - длина строки
rep movsb               ;пересылка строки
...
```

Как видим, изменилась только строка с командой пересылки. Отличие в том, что программа из листинга 12.1 может работать с цепочками элементов любой из трех размерностей: 8, 16 или 32 бита, а последний фрагмент — только с цепочками байтов. Далее, как мы и договорились раньше, чтобы не загромождать описания, мы будем рассматривать группы команд для операций-примитивов только на примере более общей команды, а вы будете понимать, что на самом деле можно использовать любую из трех команд в соответствующем контексте.

Сравнение цепочек

Команды, реализующие операцию-примитив сравнения цепочек, производят сравнение элементов цепочки-источника с элементами цепочки-приемника. Здесь ситуация с набором команд и методами работы с ними аналогична операции-примитиву пересылки цепочек. TASM предоставляет программисту четыре команды сравнения цепочек, работающие с разными размерами элементов цепочки:

- CMPS адрес_приемника,адрес_источника — сравнить строки (CoMPare String);
- CMPSB — сравнить строку байтов (CoMPare String Byte);
- CMPSW — сравнить строку слов (CoMPare String Word);
- CMPSD — сравнить строку двойных слов (CoMPare String Double word).

Команда CMPS

Синтаксис команды CMPS:

```
cmps адрес_приемника,адрес_источника
```

Здесь:

- адрес_источника определяет адрес цепочки-источника в сегменте данных, заранее загружаемый в пару регистров DS:ESI/SI;
- адрес_приемника определяет адрес цепочки-приемника, которая должна находиться в дополнительном сегменте, заранее загружаемый в пару регистров ES:EDI/DI.

Алгоритм работы команды CMPS заключается в последовательном выполнении вычитания (элемент цепочки-источника минус элемент цепочки-получателя) над очередными элементами обеих цепочек. Принцип выполнения вычитания командой CMPS такой же, как у команды сравнения CMP. Она так же, как и CMP, производит вычитание элементов, не записывая при этом результата, и устанавливает флаги ZF, SF и OF. После вычитания очередных элементов цепочек командой CMPS индексные регистры ESI/SI и EDI/DI автоматически изменяются в соответствии со значением флага DF на значение, равное размеру элемента сравниваемых цепочек. Чтобы заставить команду CMPS выполняться несколько раз, то есть произвести последовательное сравнение элементов цепочек, необходимо перед командой CMPS определить префикс повторения. С командой CMPS можно использовать префиксы повторения REP, REPE/REPZ или REPNE/REPNZ:

- REP — сравнивать элементы цепочек, пока ECX/CX>0;
- REPE или REPZ — сравнивать элементы цепочек до выполнения одного из двух условий:
  - содержимое ECX/CX равно нулю;
  - в цепочках встретились разные элементы (флаг ZF стал равен нулю);
- REPNE или REPNZ — сравнивать элементы цепочек до выполнения одного из двух условий:
  - содержимое ECX/CX равно нулю;
  - в цепочках встретились одинаковые элементы (флаг ZF стал равен единице).

Таким образом, выбор подходящего префикса позволяет организовать более гибкий поиск одинаковых или различающихся элементов цепочек командой CMPS. Критерий для выбора префикса — условие выхода из цикла. Для определения конкретного условия наиболее подходящим является способ, использующий команду условного перехода JCXZ. Ее работа заключается в анализе содержимого регистра ECX/CX, и если оно равно нулю, то управление передается на метку, указанную в качестве операнда JCXZ. Так как в регистре ECX/CX содержится счетчик повторений для цепочечной команды, имеющей любой из префиксов повторения, то, анализируя ECX/CX, можно определить условия выхода из цикла цепочечной команды. Если значение в ECX/CX не равно нулю, то это означает, что выход произошел по причине совпадения либо несовпадения очередных элементов цепочек. Существует возможность еще больше конкретизировать информацию об условии завершения операции сравнения. Сделать это можно с помощью команд условной передачи управления (табл. 12.1).

Таблица 12.1. Соответствие команд условной передачи управления условиям завершения команды CMPS

Условие завершения сравнения	Соответствующая команда условного перехода
операнд_источник > операнд_приемник	JA или JG (операнд со знаком)
операнд_источник = операнд_приемник	JE (в том числе для операнда со знаком)
операнд_источник <>операнд_приемник	JNE или JNZ (в том числе для операнда со знаком)
операнд_источник < операнд_приемник	JB или JL (операнд со знаком)
операнд_источник <= операнд_приемник	JBE или JLE (операнд со знаком)
операнд_источник >= операнд_приемник	JAE или JGE (операнд со знаком)

Как определить местоположение очередных совпавших или несовпавших элементов в цепочках? Вспомните, что после каждой итерации цепочечная команда автоматически осуществляет инкремент-декремент значения адреса в соответствующих индексных регистрах. Поэтому после выхода из цикла в этих регистрах будут находиться адреса элементов, находящихся в цепочке после (!) элементов, послуживших причиной выхода из цикла. Для получения истинного адреса этих элементов необходимо скорректировать содержимое индексных регистров, увеличив либо уменьшив значение в них на длину элемента цепочки.

В качестве примера рассмотрим программу из листинга 12.2, в которой сравнивается две строки, находящиеся в одном сегменте. Используется команда CMPS. Префикс повторения — REPE.

Листинг 12.2. Сравнение двух строк командой CMPS

```
;prg_12_2.asm
MODEL   small
STACK   256
.data
match   db  0ah,0dh,'Сравнение закончено','$'
failed  db  0ah,0dh,'Строки не совпадают','$'
string1 db  "0123456789',0ah,0dh,'$';исследуемые строки
string2 db  "9123406780','$'
.code
ASSUME  ds:@data,es:@data         ;привязка DS и ES к сегменту данных
main:
    mov ax,@data                  ;загрузка сегментных регистров
    mov ds,ax
    mov es,ax                     ;настройка ES на DS
;вывод на экран исходных строк string1 и string2
    mov ah,09h
    lea dx,string1
    int 21h
    lea dx,string2
    int 21h
;сброс флага DF - сравнение в направлении возрастания адресов
    cld
    lea si,string1                ;загрузка в si смещения string1
    lea di,string2                ;загрузка в di смещения string2
    mov cx,10                     ;длина строки для префикса repe
;сравнение строк (пока сравниваемые элементы строк равны)
```

продолжение ⇒

9 Зак. 256

Листинг 12.2 *(продолжение)*

```
<27>;выход при обнаружении несовпавшего элемента
<28>cycl:
<29>    jcxz end_hand    ;для последнего элемента
<30>repe     cmps     string1,string2
<31>    je  end_hand     ;для последнего элемента
<32>;не совпали
<33>    mov ah,09h
<34>    lea dx,failed
<35>    int. 21h             ; вывод сообщения
<36>;теперь, чтобы обработать несовпавший элемент в строке,
    ;необходимо уменьшить значения регистров si и di
<37>    dec si
<38>    dec di
<39>;сейчас в ds:si и es:di адреса несовпавших элементов
<40>;здесь вставить код по обработке несовпавшего элемента
<41>;после этого продолжить поиск в строке:
<42>    inc si
<43>    inc di
<44>    jmp cycl
<45>end_hand:
<46>    mov ah,09h           ;вывод сообщения
<47>    lea dx,match
<48>    int. 21h
<49>exit:                    ;выход
<50>    mov ax,4c00h
<51>    int 21h
<52>end main                 ;конец программы
```

В программе есть несколько требующих пояснений моментов. Это, во-первых, строки 37 и 38, в которых мы скорректировали адреса очередных элементов для получения адресов несовпавших элементов. Необходимо понимать, что если сравниваются цепочки с элементами слов или двойных слов, то нужно корректировать содержимое регистров ESI/SI и EDI/DI на 2 п. 4 байта соответственно. Во-вторых, это строки 42 и 43. Смысл их в том, что для просмотра оставшейся часта строк необходимо установить указатели на следующие элементы строк за последними несовпавшими элементами. После этого можно повторить весь процесс просмотра и обработки несовпавших элементов в оставшихся частях строк. В-третьих, это строки 28–31, с помощью которых учитываются особенности самих строк. Особое внимание следует уделить обработке последних элементов строк в предположении, что они могут не совпасть. Предлагаю вам проследить за выполнением программы в отладчике на следующих наборах входных данных:

- stringl = '0123456789' п string2 = '0123406789';
- stringl = '0123456789' и string2 = '9123406780'.

Сравнение байтов, слов и двойных слов

Так же как п в группе команд пересылки цепочки, в группе команд сравнения есть отдельные команды для сравнения цепочек байтов, слов, двойных слов — CMPSB, CMPSW и CMPSD соответственно. Для этих команд все рассуждения аналогичны тем, что были приведены при описании команд пересылки. Ассемблер преобразует команду CMPS в одну из машинных команд, CMPSB, CMPSW или CMPSD, в зависимости от размера элементов сравниваемых цепочек.

Сканирование цепочек

Команды, реализующие операцию-примитив сканирования цепочек, производят поиск некоторого значения в области памяти. Логически эта область памяти рассматривается как последовательность (цепочка) элементов фиксированной длины размером 8, 16 или 32 бита. Искомое значение предварительно должно быть помещено в один из регистров AL/AX/EAX. Выбор конкретного регистра из этих трех должен быть согласован с размером элементов цепочки, в которой осуществляется поиск. Система команд процессора предоставляет программисту четыре команды сканирования цепочки. Выбор конкретной команды определяется размером элемента:

- SCAS адрес_приемника — сканировать цепочку (SCAning String);
- SCASB — сканировать цепочку байтов (SCAning String Byte);
- SCASW — сканировать цепочку слов (SCAning String Word);
- SCASD — сканировать цепочку двойных слов (SCAning String Double Word).

Команда SCAS

Синтаксис команды SCAS:

```
scas адрес_приемника
```

Команда имеет один операнд, обозначающий местонахождение цепочки в дополнительном сегменте (адрес цепочки должен быть заранее сформирован в регистрах ES:EDI/DI). Транслятор анализирует тип идентификатора адрес_приемника, который обозначает цепочку в сегменте данных, и формирует одну из трех машинных команд, SCASB, SCASW или SCASD. Условие поиска для каждой из этих трех команд находится в строго определенном месте. Так, если цепочка описана с помощью директивы DB, то искомый элемент должен быть байтом, находиться в регистре AL, и сканирование цепочки осуществляется командой SCASB. Если цепочка описана с помощью директивы DW, то это — слово в регистре AX, и поиск ведется командой SCASW. Если цепочка описана с помощью директивы DD, то это — двойное слово в EAX, и поиск ведется командой SCASD. Принцип поиска тот же, что и в команде сравнения CMPS, то есть последовательное выполнение вычитания (содержимое регистра аккумулятора минус содержимое очередного элемента цепочки). В зависимости от результатов вычитания производится установка флагов, при этом сами операнды не изменяются. Так же как и в случае команды CMPS, с командой SCAS удобно использовать префиксы REPE/REPZ или REPNE/REPNZ.

- REPE или REPZ — если нужно организовать поиск до тех пор, пока не будет выполнено одно из двух условий:
  - будет достигнут конец цепочки (содержимое ECX/CX равно 0);
  - в цепочке встретится элемент, отличный от элемента в регистре AL/AX/EAX.
- REPNE или REPNZ — если нужно организовать поиск до тех пор, пока не будет выполнено одно из двух условий:
  - будет достигнут конец цепочки (содержимое ECX/CX равно 0);
  - в цепочке встретится элемент, совпадающий с элементом в регистре AL/AX/EAX.

Таким образом, команда SCAS с префиксом REPE/REPZ позволяет найти элемент цепочки, отличающийся по значению от заданного в аккумуляторе. Команда SCAS с префиксом REPNE/REPNZ позволяет найти элемент цепочки, совпадающий по значению с элементом в аккумуляторе. В качестве примера рассмотрим листинг 12.3, в котором реализован поиск символа в строке. В программе используется команда-примитив SCAS. Символ задается явно (строка 20). Префикс повторения — REPNE.

Листинг 12.3. Поиск символа в строке командой SCAS

```
;prg_12_3.asm
MASM
MODEL   small
STACK   256
.data
;тексты сообщений
fnd db   0ah,0dh,'Символ найден!  ',"$'
nochar  db  0ah,0dh,'Символ не найден.','$'
;строка для поиска
string  db  "Поиск символа в этой строке.",0ah,0dh,'$'
.code
ASSUME  ds:@data,es:@data
main:
    mov ax,@data
    mov ds,ax
    mov es,ax        ;настройка ES на DS
    mov ah,09h
    lea dx,string
    int 21h           ;вывод сообщения string
    mov al,'a'        ;символ для поиска - "а"(кириллица)
    cld               ;сброс флага df
    lea di,string    ;загрузка в es:di смещения строки
    mov cx,28         ;для префикса repne - длина строки
;поиск в строке (пока искомый символ и символ в строке не совпадут)
;выход при первом совпадении
repne   scas string
    je  found         ;если равны - переход на обработку,
failed:              ;иначе - выполняем некоторые действия
;вывод сообщения о том, что символ не найден
    mov ah,09h
    lea dx,nochar
    int 21h           ;вывод сообщения nochar
    jmp exit          ;на выход
found:               ;совпали
    mov ah,09h
    lea dx,fnd
    int 21h           ;вывод сообщения fnd
;теперь, чтобы узнать место, где совпал элемент в строке,
;необходимо уменьшить значение в регистре di и вставить нужный обработчик
;   dec di
;... вставьте обработчик
exit:                ;выход
    mov ax,4c00h
    int 21h
end main
```

Сканирование строки байтов, слов, двойных слов

Система команд процессора, так же как в случае операций-примитивов пересылки и сравнения, предоставляет команды сканирования, явно указывающие размер элемента цепочки — SCASB, SCASW или SCASD. Помните, что даже если вы этого не делаете, то ассемблер все равно преобразует команду SCAS в одну из этих трех машинных команд.

Загрузка элемента цепочки в аккумулятор

Операция-примитив загрузки элемента цепочки в аккумулятор позволяет извлечь элемент цепочки и поместить его в регистр-аккумулятор AL, AX или EAX. Эту операцию удобно использовать вместе с поиском (сканированием) с тем, чтобы, найдя нужный элемент, извлечь его (например, для изменения). Возможный размер извлекаемого элемента определяется применяемой командой. Программист может использовать четыре команды загрузки элемента цепочки в аккумулятор, работающие с элементами разного размера:

- LODS адрес_источника — загрузить элемент из цепочки (LOaD String) в регистр-аккумулятор AL/AX/EAX;
- LODSB — загрузить байт из цепочки (LOaD String Byte) в регистр AL;
- LODSW — загрузить слово из цепочки (LOaD String Word) в регистр AX;
- LODSD — загрузить двойное слово (LOaD String Double Word) из цепочки в регистр EAX.

Рассмотрим работу этих команд на примере команды LODS.

Команда LODS

Синтаксис команды LODS:

```
lods адрес_источника
```

Команда имеет один операнд, обозначающий строку в основном сегменте данных. Работа команды заключается в том, чтобы извлечь элемент из цепочки по адресу, соответствующему содержимому пары регистров DS:ESI/SI, и поместить его в регистр EAX/AX/AL. При этом содержимое ESI/SI подвергается инкременту или декременту (в зависимости от состояния флага DF) на величину, равную размеру элемента. Эту команду удобно использовать после команды SCAS, локализующей местоположение искомого элемента в цепочке. Префикс повторения в этой команде может и не понадобиться — все зависит от логики программы.

В качестве примера рассмотрим листинг 12.4, в котором командой CMPS сравниваются две цепочки байтов в памяти (string1 и string2) и первый не совпавший байт из string2 помещается в регистр AL. Для загрузки этого байта в регистр-аккумулятор AL используется команда LODS. Префикса повторения в команде LODS нет, так как он попросту не нужен.

Листинг 12.4. Использование команды LODS для загрузки байта в регистр AL

```
;prg_11_4.asm
MASM
MODEL   small
STACK   256
.data
;строки для сравнения
string1 db   "Поиск символа в этой строке.",0ah,0dh,'$'
string2 db   "Поиск символа не в этой строке.",0ah,0dh,'$'
mes_eq  db   "Строки совпадают.",0ah,0dh,'$'
fnd db  "Несовпавший элемент в регистре al",0ah,0dh,'$'
.code
;привязка ds и es к сегменту данных
assume ds:@data,es:@data
```

продолжение ↗

Листинг 12.4 *(продолжение)*

```
main:
    mov ax,@data;загрузка сегментных регистров
    mov ds,ax
    mov es,ax           ;настройка es на ds
    mov ah,09h
    lea dx,string1
    int 21h             ;вывод string1
    lea dx,string2
    int 21h             ;вывод string2
    cld                 ;сброс флага df
    lea di,string1      ;загрузка в es:di смещения строки string1
    lea si,string2      ;загрузка в ds:si смещения строки string2
    mov cx,29           ;для префикса repe -- длина строки
                        ;поиск в строке (пока нужный символ и символ
                        ;в строке не равны)
;выход - при первом не совпавшем
repe    cmps string1,string2
    jcxz eql            ;если равны - переход на eql
    jmp no_eq           ;если не равны - переход на no_eq
eql:                    ;выводим сообщение о совпадении строк
    mov ah,09h
    lea dx,mes_eq
    int 21h             ;вывод сообщения mes_eq
    jmp exit            ;на выход
no_eq:                  ;обработка несовпадения элементов
    mov ah,09h
    lea dx,fnd
    int 21h             ;вывод сообщения fnd
;теперь, чтобы извлечь несовпавший элемент из строки
;в регистр-аккумулятор,
;уменьшаем значение регистра si и тем самым перемещаемся
;к действительной позиции элемента в строке
    dec si              ;команда lods использует ds:si-адресацию
;теперь ds:si указывает на позицию в string2
    lods string2        ;загрузим элемент из строки в AL
;нетрудно догадаться, что в нашем примере это символ - "н"
exit:                   ;выход
    mov ax,4c00h
    int 21h
end main
```

Загрузка в регистр AL/AX/EAX байтов, слов, двойных слов

Команды загрузки байта в регистр AL (LODSB), слова — в регистр AX (LODSW), двойного слова в регистр EAX (LODSD) аналогичны другим цепочечным командам. Они являются вариантами команды LODS. Каждая из этих команд работает с цепочками из элементов определенного размера. Предварительно вы должны загрузить значение длины цепочки и ее адрес в регистры ECX/CX и DS:ESI/SI.

Перенос элемента из аккумулятора в цепочку

Операция-примитив переноса элемента из аккумулятора в цепочку позволяет произвести действие, обратное действию команды LODS, то есть сохранить значение из регистра-аккумулятора в элементе цепочки. Эту операцию удобно использовать

вместе с операциями поиска (сканирования) SCANS и загрузки LODS с тем, чтобы, найдя нужный элемент, извлечь его в регистр и записать на его место новое значение. Команды, поддерживающие эту операцию-примитив, могут работать с: элементами размером 8, 16 или 32 бита. TASM предоставляет программисту четыре команды сохранения элемента цепочки из регистра-аккумулятора, работающие с элементами разного размера:

- STOS адрес_приемника — сохранить в цепочке элемент (STOre String) из регистра-аккумулятора AL/AX/EAX;
- STOSB — сохранить в цепочке байт (STOre String Byte) из регистра AL;
- STOSW — сохранить в цепочке слово (STOre String Word) из регистра AX;
- STOSD — сохранить в цепочке двойное слово (STOre Siring Double Word) из регистра EAX.

Команда STOS

Синтаксис команды STOS:

```
stos адрес_приемника
```

Команда имеет один операнд адрес_приемника, адресующий цепочку в дополнительном сегменте данных. Команда пересылает элемент из аккумулятора (регистра EAX/AX/AL)в элемент цепочки по адресу, соответствующему содержимому пары регистров ES:EDI/DI. При этом содержимое EDI/DI подвергается инкременту или декременту (в зависимости от состояния флага DF) па величину, равную размеру элемента цепочки.

Префикс повторения в этой команде может и не понадобиться — все зависит от логики программы. Например, если использовать префикс повторения REP, то можно применить команду для инициализации области памяти некоторым фиксированным значением.

В качестве примера рассмотрим листинг 12.5, в котором производится замена в строке всех символов «а» символами, вводимыми с клавиатуры.

Листинг 12.5. Замена командой STOS символа в строке символом, вводимым с клавиатуры

```
;prg_12_5.asm
MASM
MODEL   small
STACK   256
.data
;сообщения
fnd db   0ah,0dh,'Символ найден','$'
nochar  db  0ah,0dh,'Символ не найден.','$'
mes1    db  0ah,0dh,'Исходная строка:','$'
string  db   "Поиск символа в этой строке.",0ah,0dh,'$'  ;строка для поиска
mes2    db  0ah,0dh,'Введите символ, на который следует заменить найденный'
    db  0ah,0dh,'$'
mes3    db  0ah,0dh,'Новая строка: ","$'
.code
    assume  ds:@data,es:@data     ;привязка ds и es
                                  ;к сегменту данных
main:                             ;точка входа в программу
    mov ax,@data                  ;загрузка сегментных регистров
```

продолжение ⇨

Листинг 12.5 *(продолжение)*

```
    mov ds,ax
    mov es,ax          ;настройка es на ds
    mov ah,09h
    lea dx,mes1
    int 21h            ;вывод сообщения mes1
    lea dx,string
    int 21h            ;вывод string
    cld                ;сброс флага df
    lea di,string      ;загрузка в di смещения string
    mov cx,28          ;для префикса repne - длина строки
;поиск в строке string до тех пор, пока
;символ в al и очередной символ в строке
;не равны: выход — при первом совпадении
cycl:
    mov al,'a'         ;символ для поиска - "a"(кириллица)
repne  scas string
    je  found          ;если элемент найден, то переход на found
failed:                ;иначе, если не найден, то вывод сообщения nochar
    mov ah,09h
    lea dx,nochar
    int 21h
    jmp exit           ;переход на выход
found:
    mov ah,09h
    lea dx,fnd
    int 21h            ;вывод сообщения об обнаружении символа
;корректируем di для получения значения
;действительной позиции совпавшего элемента
;в строке и регистре al
    dec di
new_char:              ;блок замены символа
    mov ah,09h
    lea dx,mes2
    int 21h            ;вывод сообщения mes2
;ввод символа с клавиатуры
    mov ah,01h
    int 21h            ;в al - введенный символ
    stos    string     ;сохраним введенный символ
                       ; (из al) в строке
                       ;string в позиции старого символа
    mov ah,09h
    lea dx,mes3
    int 21h            ;вывод сообщения mes3
    lea dx,string
    int 21h            ;вывод сообщения string
;переход на поиск следующего символа "a" в строке
    inc di             ;указатель в строке string на символ,
                       ;следующий после совпавшего,
    jmp cycl           ;на продолжение просмотра string
exit:                  ;выход
    mov ax,4c00h
    int 21h
end  main              ;конец программы
```

Сохранение в цепочке байта, слова, двойного слова из регистра AL/AX/EAX

Команды STOSB, STOSW и STOSD, аналогично другим цепочечным операциям, являются вариантами команды STOS. Каждая из этих команд работает с цепочками из элементов определенного размера. Предварительно необходимо загрузить значение длины цепочки и ее адрес в регистры ECX/CX и ES:EDI/DI.

Работа с портами ввода-вывода

Описанные далее две команды появились впервые в системе команд процессора i386. Они позволяют организовать эффективную передачу данных между портами ввода-вывода и цепочками в памяти. Следует отметить, что эти две команды позволяют достичь более высокой скорости передачи данных по сравнению с той скоростью, которую может обеспечить контроллер DMA (Direct Memory Access — прямой доступ к памяти).

Ввод элемента цепочки из порта ввода-вывода

Операция ввода элемента цепочки из порта ввода-вывода реализуется командой INS (Input String), имеющей следующий формат:

```
ins адрес_приемника,номер_порта
```

Эта команда вводит элемент из порта, номер которого находится в регистре DX, в элемент цепочки, адрес памяти которого определяется операндом адрес_приемника. Несмотря на то, что цепочка, в которую вводится элемент, адресуется указанием этого операнда, ее адрес должен быть явно сформирован в паре регистров ES:EDI/DI. Размер элементов цепочки должен быть согласован с размером порта — он определяется директивой резервирования памяти, с помощью которой выделяется память для размещения элементов цепочки. После пересылки команда INS производит коррекцию содержимого регистра EDI/DI на величину, равную размеру элемента, участвовавшего в операции пересылки. Как обычно, при работе цепочечных команд учитывается состояние флага DF.

Подобно командам, реализующим рассмотренные ранее цепочечные операции-примитивы, транслятор преобразует команду INS в одну из трех машинных команд без операндов, работающих с цепочками элементов определенного размера:

- INSB (INput String Byte) — ввести из порта цепочку байтов;
- INSW (INput String Word) — ввести из порта цепочку слов;
- INSD (INput String Double Word) — ввести из порта цепочку двойных слов.

К примеру, введем из порта 5000h 10 байтов в область памяти pole:

```
.data
pole   db  10 dup (" ")
.code
...
    push ds
    pop es  ;настройка es на ds
    mov dx,5000h
    lea di,pole
    mov cx,10
rep insb
...
```

Вывод элемента цепочки в порт ввода-вывода

Операция вывода элемента цепочки в порт ввода-вывода реализуется командой OUTS (Output String), имеющей следующий формат:

```
outs номер_порта,адрес_источника
```

Эта команда выводит элемент цепочки в порт, номер которого находится в регистре DX. Адрес элемента цепочки определяется операндом адрес_источника.

Несмотря на то, что цепочка, из которой выводится элемент, адресуется указанием этого операнда, значение адреса должно быть явно сформировано в паре регистров DS:ESI/SI. Размер структурных элементов цепочки должен быть согласован с размером порта — он определяется директивой резервирования памяти, с помощью которой выделяется память для размещения элементов цепочки. После пересылки команда OUTS производит коррекцию содержимого регистра ESI/SI на величину, равную размеру элемента цепочки, участвовавшего в операции пересылки. При этом, как обычно, учитывается состояние флага DF.

Подобно команде INS транслятор преобразует команду OUTS в одну из трех машинных команд без операндов, работающих с цепочками элементов определенного размера:

- OUTSB (OUTput String Byte) – вывести цепочку байтов в порт ввода-вывода;
- OUTSW (OUTtput String Word) — вывести цепочку слов в порт ввода-вывода;
- OUTSD (OUTput String Double Word) — вывести цепочку двойных слов в порт ввода-вывода.

В качестве примера рассмотрим фрагмент программы, которая выводит последовательность символов в порт ввода-вывода с номером 378 (lpt1), соответствующий принтеру:

```
.data
str_pech    db   "Текст для печати"
.code
...
   mov dx,378h
   lea di,str_pech
   mov cx,16
rep outsb
...
```

Для организации работы с портами недостаточно знать их номера и назначение. Не менее важно знать и понимать алгоритмы их работы. Эти сведения можно найти в документации на устройство (но, к сожалению, далеко не всегда).

Реализации более сложных алгоритмов (поиска), основанные на работе с цепочечными командами, приведены в [8].

Итоги

- Система команд процессора имеет очень интересную группу команд, позволяющих производить действия над блоками элементов до 64 Кбайт или 4 Гбайт в зависимости от установленной разрядности адреса — use16 или use32.
- Блоки элементов логически могут представлять собой последовательности элементов с любыми значениями, хранящимися в памяти в виде двоичных кодов. Единственное ограничение состоит в том, что размеры элементов в этих блоках памяти фиксированы значением 8, 16 или 32 бита.
- Команды обработки строк обеспечивают возможность выполнения семи операций-примитивов, обрабатывающих цепочки поэлементно.
- Каждая операция-примитив представлена тремя разными машинными командами и одной псевдокомандой, которая преобразуется транслятором в одну из

трех упомянутых ранее машинных команд. Вариант преобразования определяется типом операндов в команде.

- Процессор всегда предполагает, что строка-приемник находится в дополнительном сегменте (адресуемом посредством сегментного регистра ES), а строка-источник — в сегменте данных (адресуемом посредством сегментного регистра DS).
- Процессор адресует строку-приемник через регистр EDI/DI, а строку-источник — через регистр ESI/SI.
- Допускается переопределять сегмент для строки-источника, для строки-приемника этого делать нельзя.
- Особенность работы цепочечных команд состоит в том, что они автоматически выполняют приращение или уменьшение содержимого регистров EDI/DI и ESI/SI в зависимости от используемой цепочечной команды. Что именно происходит с этими регистрами, определяется состоянием флага DF, которым управляют команды CLD и STD. Значение, на которое изменяется содержимое индексных регистров, определяется типом элементов строки или кодом операции цепочечной команды.

Глава 13
Сложные структуры данных

- **Понятие сложного типа данных в ассемблере**
- **Средства ассемблера для создания и обработки сложных структур данных**
- **Массивы**
- **Структуры**
- **Объединения**
- **Записи**

В предыдущих главах при разработке программ мы использовали данные двух типов.

- Непосредственные данные, представляющие собой числовые или символьные значения и являющиеся частью команды. Непосредственные данные формируются программистом в процессе написания программы для конкретной команды ассемблера.
- Данные, описываемые с помощью ограниченного набора директив резервирования памяти, позволяют выполнять самые элементарные операции по размещению и инициализации числовой и символьной информации. При обработке этих директив ассемблер сохраняет в своей таблице символов информацию о местоположении данных (значения сегментной составляющей адреса и смещения) и типе данных, то есть единицах памяти, выделяемых для размещения данных в соответствии с директивой резервирования и инициализации данных.

Эти два типа данных являются *элементарными,* или *базовыми;* работа с ними поддерживается на уровне системы команд процессора. Используя данные этих типов, можно формализовать и запрограммировать практически любую задачу. Но насколько это будет удобно — вот вопрос.

Обработка информации в общем случае — процесс очень сложный. Это косвенно подтверждает популярность языков высокого уровня. Одно из несомненных достоинств языков высокого уровня — поддержка развитых структур данных. При их использовании программист освобождается от решения конкретных проблем, связанных с представлением числовых или символьных данных, и получает возможность оперировать информацией, структура которой в большей степени отражает особенности предметной области решаемой задачи. В то же самое время, чем выше уровень такого абстрагирования от конкретного представления данных в компьютере, тем большая нагрузка должна ложиться на компилятор для создания действительно эффективного кода. Ведь нам уже известно, что в конечном итоге все написанное на языке высокого уровня в компьютере будет представлено на уровне машинных команд, работающих только с базовыми типами данных. Таким образом, самая эффективная программа — программа, написанная в машинных кодах, но писать сегодня большую программу в машинных кодах — занятие довольно бессмысленное.

С целью облегчения разработки программ в язык ассемблера на уровне его директив была введена поддержка нескольких сложных типов данных. Это позволило несколько сгладить различия между языками высокого уровня и ассемблером. У программиста появилась возможность сочетать достоинства языка ассемблера и языков высокого уровня (в направлении абстрагирования от конкретного представления данных), что в итоге повышает эффективность конечной программы.

TASM поддерживает следующие сложные типы данных:

- массивы;
- структуры;
- объединения;
- записи.

Разберемся более подробно с тем, как определить данные этих типов в программе и организовать работу с ними.

Массивы

Дадим формальное определение: *массив* — структурированный тип данных, состоящий из некоторого числа элементов одного типа.

Для того чтобы разобраться в возможностях и особенностях обработки массивов в программах на ассемблере, нужно ответить на следующие вопросы.

- Как описать массив в программе?
- Как инициализировать массив, то есть как задать начальные значения его элементов?
- Как организовать доступ к элементам массива?
- Как организовать выполнение типовых операций с массивами?

Описание и инициализация массива в программе

Специальных средств описания массивов в программах ассемблера, конечно, нет. Чтобы использовать массив в программе, его нужно смоделировать одним из перечисленных далее способов.

- Можно перечислить элементы массива в поле операндов одной из директив описания данных. При перечислении элементы разделяются запятыми. Например,

```
;массив из 5 элементов. Размер каждого элемента 4 байта:
mas dd  1,2,3,4,5
```

- Можно использовать оператор повторения DUP. К примеру,

```
;массив из 5 нулевых элементов. Размер каждого элемента 2 байта:
mas dw   5 dup  (0)
```

Такой способ определения используется для резервирования памяти с целью размещения и инициализации элементов массива.

- Можно использовать директивы LABEL и REPT. Пара этих директив позволяет облегчить описание больших массивов в памяти и повысить наглядность такого описания. Директива REPT относится к макросредствам языка ассемблера и вызывает повторение указанное число раз строк, заключенных между директивой и строкой ENDM. К примеру, определим массив байтов в области памяти, обозначенной идентификатором mas_b. В данном случае директива LABEL определяет символическое имя mas_b аналогично тому, как это делают директивы резервирования и инициализации памяти. Достоинство директивы LABEL — в том, что она не резервирует память, а лишь определяет характеристики объекта. В данном случае объект — это ячейка памяти. Используя несколько директив LABEL, записанных одна за другой, можно присвоить одной и той же области памяти разные имена и типы, что и сделано в следующем фрагменте:

```
...
n=0
...
mas_b   label   byte
mas_w   label   word
rept    4
    dw  0f1f0h
endm
```

В результате в памяти будет создана последовательность из четырех слов f1f0. Эту последовательность можно трактовать как массив байтов или слов в зависимости от того, какое имя области мы будем использовать в программе — mas_b или mas_w.

- Чтобы инициализировать значениями область памяти и впоследствии трактовать ее как массив, можно использовать цикл. Посмотрим па примере листинга 13.1, каким образом это делается.

Листинг 13.1. Инициализация массива в цикле

```
;prg_13_1.asm
MASM
MODEL   small
STACK   256
.data
mes db  0ah,0dh,'Массив- ',"$'
mas db  10 dup (?)        ;исходный массив
:   db  0
.code
main:
```

```
        mov ax,@data
        mov ds,ax
        xor ax,ax               ;обнуление ax
        mov cx,10               ;значение счетчика цикла в cx
        mov si,0                ;индекс начального элемента в cx
go:                             ;цикл инициализации
        mov bh,i                ;i в bh
        mov mas[si],bh          ;запись в массив i
        inc i                   ;инкремент i
        inc si                  ;продвижение к следующему элементу массива
        loop go                 ;повторить цикл
;вывод на экран получившегося массива
        mov cx,10
        mov si,0
        mov ah,09h
        lea dx,roes
        int 21h
show:
        mov ah,02h              ;функция вывода значения из al на экран
        mov dl,mas[si]
        add dl,30h              ;преобразование числа в символ
        int 21h
        inc si
        loop show
exit:
        mov ax,4c00h            ;стандартный выход
        int 21h
end     main                    ;конец программы
```

Доступ к элементам массива

При работе с массивами необходимо четко представлять себе, что все элементы массива располагаются в памяти компьютера последовательно. Само по себе такое расположение ничего не говорит о назначении и порядке использования этих элементов. И только лишь программист с, помощью составленного им алгоритма обработки определяет, как нужно трактовать последовательность байтов, составляющих массив. Так, одну и ту же область памяти можно трактовать одновременно и как одномерный, и как двухмерный массив. Все зависит только от алгоритма обработки этих данных в конкретной программе. Сами по себе данные не несут никакой информации о своем «смысловом», или логическом, типе. Помните об этом принципиальном моменте.

Те же соображения можно распространить и на индексы элементов массива. Ассемблер не подозревает ни об их существовании, ни об их численных смысловых значениях. Для того чтобы локализовать определенный элемент массива, к его имени нужно добавить индекс. Так как мы моделируем массив, то должны позаботиться и о моделировании индекса. В языке ассемблера индексы массивов — это обычные адреса, но с ними работают особым образом. Другими словами, когда при программировании на ассемблере мы говорим об индексе, то, скорее, подразумеваем под этим не номер элемента в массиве, а некоторый адрес. Давайте еще раз обратимся к описанию массива. К примеру, пусть в программе статически определена последовательность данных:

```
mas dw 0,1,2,3,4,5
```

Пусть эта последовательность чисел трактуется как одномерный массив. Размерность каждого элемента определяется директивой DW, то есть она равна двум байтам. Чтобы получить доступ к третьему элементу, нужно к адресу массива при-

бавить 6. Нумерация элементов массива в ассемблере начинается с нуля. То есть в нашем случае речь, фактически, идет о 4-м элементе массива — 3, но об этом знает только программист; процессору в данном случае все равно — ему нужен только адрес. В общем случае для получения адреса элемента в массиве необходимо начальный (базовый) адрес массива сложить с произведением индекса (номер элемента минус единица) этого элемента на размер элемента массива:

база + (индекс • размер элемента).

Архитектура процессора предоставляет довольно удобные программно-аппаратные средства для работы с массивами. К ним относятся базовые и индексные регистры, позволяющие реализовать несколько режимов адресации данных. Используя данные режимы адресации, можно организовать эффективную работу с массивами в памяти. Вспомним эти режимы.

- Индексная адресация со смещением — режим адресации, при котором эффективный адрес формируется из двух компонентов:
  - постоянный (базовый) компонент формируется указанием прямого адреса массива в виде имени идентификатора, обозначающего начало массива;
  - переменный (индексный) компонент формируется указанием имени индексного регистра.

  К примеру,

```
mas dw  0,1,2,3,4,5
...
    mov si ,4
;поместить 3-й элемент массива mas в регистр ax:
    mov ax,mas[si]
```

- Базовая индексная адресация со смещением — режим адресации, при котором эффективный адрес формируется максимум из трех компонентов:
  - в качестве постоянного (необязательного) компонента может выступать прямой адрес массива в виде имени идентификатора, обозначающего начало массива, или непосредственного значения;
  - переменный (базовый) компонент формируется указанием имени базового регистра;
  - переменный (индексный) компонент формируется указанием имени индексного регистра.

  Этот вид адресации удобно использовать при обработке двухмерных массивов. Пример использования этой адресации мы рассмотрим далее при изучении особенностей работы с двухмерными массивами.

Напомним, что в качестве базового регистра может использоваться любой из восьми регистров общего назначения. В качестве индексного регистра также можно использовать любой регистр общего назначения, за исключением ESP/SP.

Процессор позволяет *масштабировать индекс*. Это означает, что если указать после имени индексного регистра символ звездочки (*) с последующей цифрой 2, 4 или 8, то содержимое индексного регистра будет умножаться на 2, 4 или 8, то есть масштабироваться. Применение масштабирования облегчает работу с массивами, которые имеют размер элементов, равный 2, 4 или 8 байт, так как процессор сам

производит коррекцию индекса для получения адреса очередного элемента массива. Нам нужно лишь загрузить в индексный регистр значение требуемого индекса (считая от 0). Кстати сказать, возможность масштабирования появилась в процессорах Intel, начиная с модели i486. По этой причине в рассматриваемом далее примере программы стоит директива **.486**. Ее назначение, как и ранее использовавшейся директивы .386, — в том, чтобы указать ассемблеру на необходимость учета дополнительных возможностей новых процессоров при формировании машинных команд.

В качестве примера масштабирования рассмотрим листинг 13.2, в котором просматривается массив, состоящий из слов, и производится сравнение этих элементов с нулем. Выводится соответствующее сообщение.

Листинг 13.2. Просмотр массива слов с использованием масштабирования

```
;prg_13_2.asm
;tlink /v /3 prg_12_2.obj
MASM
MODEL   small
STACK   256
.data                           ;начало сегмента данных
;тексты сообщений:
mes1    db   "не равен 0!$",0ah,0dh
raes2 db    "равен  0!$",0ah,0dh
mes3    db  0ah,0dh,'Элемент $'
mas dw  2,7,0,0,1,9,3,6,0,8  ;исходный массив
.code
.486                            ;это обязательно
main:
    mov ax,@data
    mov ds,ax                   ;связка ds с сегментом данных
    xor ax,ax                   ;обнуление ax
prepare:
    mov cx,10                   ;значение счетчика цикла в cx
    mov esi,0                   ;индекс в esi
compare:
    mov dx,mas[esi*2]           ;первый элемент массива в dx
    cmp dx,0                    ;сравнение dx с 0
    je  equal                   ;переход, если равно
not_equal:                      ;не равно
    mov ah,09h                  ;вывод сообщения на экран
    lea dx,mes3
    int 21h
    mov ah,02h                  ;вывод номера элемента массива на экран
    mov dx,si
    add dl,30h
    int 21h
    mov ah,09h
    lea dx,mes1
    int 21h
    inc esi                     ;на следующий элемент
    dec cx                      ;условие для выхода из цикла
    jcxz exit                   ;cx=0? Если да - на выход
    jmp compare                 ;нет - повторить цикл
equal:                          ;равно 0
    mov ah,09h                  ;вывод сообщения mes3 на экран
    lea dx,mes3
    int 21h
    mov ah,02h
    mov dx,si
    add dl,30h
```

продолжение ➭

Листинг 13.2 *(продолжение)*

```
    int 21h
    mov ah,09h              ;вывод сообщения mes2 на экран
    lea dx,mes2
    int 21h
    inc esi                 ;на следующий элемент
    dec cx                  ;все элементы обработаны?
    jcxz exit
    jmp compare
exit:
    mov ax,4c00h            ;стандартный выход
    int 21h
end main                    ;конец программы
```

Еще несколько слов о соглашениях.

- Если для описания адреса используется только один регистр, то речь идет о базовой адресации, и этот регистр рассматривается как базовый:

```
;переслать байт из области данных,
;адрес которой находится в регистре ebx:
    mov al,[ebx]
```

- Если для задания адреса в команде используется прямая адресация (в виде идентификатора) в сочетании с одним регистром, то речь идет об индексной адресации. Регистр считается индексным, и поэтому для получения адреса нужного элемента массива можно выполнить масштабирование:

```
    add eax,mas[ebx*4]
;сложить содержимое eax с двойным словом
;в памяти по адресу mas + (ebx)*4
```

- Если для описания адреса используются два регистра, то речь идет о базово-индексной адресации. Левый регистр рассматривается как базовый, правый — как индексный. В общем случае это не принципиально, но если масштабирование применяется к одному из регистров, то он всегда является индексным. Однако лучше придерживаться определенных соглашений. Помните, что применение регистров EBP/BP и ESP/SP по умолчанию подразумевает, что сегментная составляющая адреса находится в регистре SS.

Заметим, что базово-индексную адресацию не возбраняется сочетать с прямой адресацией или указанием непосредственного значения. Адрес тогда будет формироваться как сумма всех компонентов.

К примеру,

```
    mov ax,mas[ebx][ecx*2]
;адрес операнда равен [mas+(ebx)+(ecx)*2]
...
    sub dx,[ebx+8][ecx*4]
;адрес операнда равен [(ebx)+8+(ecx)*4]
```

Следует заметить, что масштабирование эффективно лишь тогда, когда размерность элементов массива равна 2, 4 или 8 байт. Если же размерность элементов другая, то организовывать обращение к элементам массива нужно обычным способом, как описано ранее.

Рассмотрим пример работы с массивом из пяти трехбайтовых элементов (листинг 13.3). Младший байт в каждом из этих элементов представляет собой некий счетчик, а старшие два байта — что-то еще, для нас не имеющее никакого значе-

ния. Необходимо последовательно обработать элементы данного массива, увеличив значения счетчиков на единицу.

Листинг 13.3. Обработка массива элементов с нечетной длиной

```
    ;prg_13_3.asm
    MASM
    MODEL   small           ;модель памяти
    STACK   256             ;размер стека
    .data                   ;начало сегмента данных
    N=5                     ;количество элементов массива
    mas db  5 dup (3 dup  (0))
    .code                   ;сегмент кода
main:                       ;точка входа в программу
        mov  ax,@data
        mov  ds,ax
        xor  ax,ax          ;обнуление ax
        mov  si,0           ;0 в si
        mov  cx,N           ;N в cx
go:
        mov  dl,mas[si]     ;первый байт поля в dl
        inc  dl             ;увеличение dl на 1 (по условию)
        mov  mas[si],dl     ;заслать обратно в массив
        add  si,3           ;сдвиг на следующий элемент массива
        loop go             ;повтор цикла
        mov  si,0           ;подготовка к выводу на экран
        mov  cx,N
show:                       ;вывод на экран содержимого
                            ;первых байтов полей
        mov  dl,mas[si]
        add  si,3
        add  dl,30h
        mov  ah,02h
        int  21h
        loop show
exit:
        mov  ax,4c00h;стандартный выход
        int  21h
end     main        ;конец программы
```

Двухмерные массивы

С представлением одномерных массивов в программе на ассемблере и организацией их обработки все достаточно просто. А как быть, если программа должна обрабатывать двухмерный массив? Все проблемы возникают по-прежнему из-за того, что специальных средств для описания такого типа данных в ассемблере нет. Двухмерный массив нужно моделировать. На описании самих данных это почти никак не отражается — память под массив выделяется с помощью директив резервирования *и* инициализации памяти. Непосредственно моделирование обработки массива производится в сегменте кода, где программист, описывая алгоритм обработки на ассемблере, определяет, что некоторую область памяти необходимо трактовать как двухмерный массив. При этом вы вольны в выборе того, как понимать расположение элементов двухмерного массива в памяти: по строкам или по столбцам. Если последовательность однотипных элементов в памяти трактуется как двухмерный массив, расположенный по строкам, то адрес элемента (i, j) вычисляется по формуле

$$(\text{база} + (\text{количество_элементов_в_строке} \cdot i + j) \cdot \text{размер_элемента}).$$

Здесь i = 0...n–1 — номер строки, aj = 0...m–1 — номер столбца. Например, пусть имеется массив чисел (размером в 1 байт) mas(i, j) с размерностью 4 • 4 (г = 0...3, j = 0...3):

23040567
05060799
67080923
87090008

В памяти элементы этого массива будут расположены в следующей последовательности:

23 04 05 67 05 06 07 99 67 08 09 23 87 09 00 08

Если мы хотим трактовать эту последовательность как двухмерный массив, приведенный раньше, и извлечь, например, элемент mas(2, 3) = 23, то, проведя нехитрый подсчет, убедимся в правильности наших рассуждений:

Эффективный адрес mas(2, 3) = mas + (4 • 2 + 3) • 1= mas + 22.

Посмотрите на представление массива в памяти и убедитесь, что по этому смещению действительно находится нужный элемент массива.

Логично организовать адресацию двухмерного массива, используя рассмотренную нами ранее базово-индексную адресацию. При этом возможны два основных варианта выбора компонентов для формирования эффективного адреса:

* сочетание прямого адреса как базового компонента адреса и двух индексных регистров для хранения индексов:

  ```
  mov ax,mas[ebx][esi]
  ```

* сочетание двух индексных регистров, один из которых является и базовым, и индексным одновременно, а другой — только индексным:

  ```
  mov ax,[ebx][esi]
  ```

В программе это будет выглядеть примерно так:

```
;Фрагмент программы выборки элемента
;массива mas(2,3) и его обнуления
.data
mas db 23,4,5,67,5,6,7,99,67,8,9,23,87,9,0,8
i=2
]=3
el_size=1
.code
...
.386
    movesi,4*el_size*i
    movdi,j*el_size
    mov al,mas[esi][edi];в al элемент mas(2,3)
...
```

В качестве законченного примера рассмотрим программу поиска элемента в двухмерном массиве чисел (листинг 13.4). Элементы массива заданы статически.

Листинг 13.4. Поиск элемента в двухмерном массиве

```
;prg_13_4.asm
    MASM
    MODEL   small
    STACK   256
    .data
```

```
    ;матрица размером 2x5 - если ее не инициализировать,
    ;то для наглядности она может быть описана так:
    ;array dw 2 DUP (5 DUP (?))
    ;но мы ее инициализируем:
    array   dw  1,2,3,3,5,6,7,3,9,0
    ;логически это будет выглядеть так:
    ;array= {1  2}
    ;       {3  3}
    ;       {5  6}
    ;       {7  3}
    ;       {9  0}
        size_elem=2             ;размер элемента
        elemdw  3               ;элемент для поиска
    failed  db  0ah,0dh,'Нет такого элемента в массиве! ','$'
    success db 0ah,0dh,'Такой элемент в массиве присутствует ',"$'
    foundtime   db ?            ;количество найденных элементов
    fnd db  ' раз(а)',0ah,0dh,'$'
    .code
main:
        mov ax,@data
        mov ds,ax
        xor ax,ax
        mov si,0                ;si=столбцы в матрице
        mov bx,0                ;bx=строки в матрице
        mov cx,5                ;число для внешнего цикла (по строкам)
external:                       ;внешний цикл по строкам
        push cx                 ;сохранение в стеке счетчика внешнего цикла
        mov cx,2                ;число для внутреннего цикла (по столбцам)
        mov si,0
iternal:                        ;внутренний цикл по строкам
        mov ax,array[bx][si];в ах первый (очередной) элемент матрицы
        add si,size_elem        ;передвижение на следующий элемент в строке
;сравниваем содержимое текущего элемента в ах с искомым элементом:
        cmp ax,elem
;если текущий совпал с искомым, то переход на here для обработки,
;иначе - цикл продолжения поиска
        jne $+6
        inc foundtime           ;увеличиваем счетчик совпавших
        loop iternal
move_next:                      ;продвижение в матрице
        pop cx                  ;восстанавливаем CX из стека (5)
        add bx,size_elem*2      ;передвигаемся на следующую строку
        loop external           ;цикл (внешний)
        cmp foundtime,0h        ;сравнение числа совпавших с 0
        ja  eql                 ;если больше 0, то переход
not_equal:                      ;нет элементов, совпавших с искомым
        mov ah,09h              ;вывод сообщения на экран
        mov dx,offset failed
        int 21h
        jmp exit                ;на выход
eql:                            ;есть элементы, совпавшие с искомым
        mov ah,09h              ;вывод сообщений на экран
        mov dx,offset success
        int 21h
        mov ah,02h
        mov dl,foundtime
        add dl,30h
        int 21h
        mov ah,09h
        mov dx,offset fnd
        int 21h
exit:                           ;выход
        mov ax,4c00h            ;стандартное завершение программы
        int 21h
end     main                    ;конец программы
```

Анализируя работу программы, не забывайте, что при написании программ на языке ассемблера нумерацию элементов массива удобнее производить с 0. При поиске определенного элемента массив просматривается от начала и до конца. Программа сохраняет в поле **foundtime** количество вхождений искомого элемента в массив. В качестве индексных используются регистры SI и BX.

Типовые операции с массивами

Для демонстрации основных приемов работы с массивами лучше всего подходят программы поиска или сортировки. Программа поиска была приведена ранее. Теперь рассмотрим одну из программ, выполняющих сортировку массива по возрастанию (листинг 13.5).

Листинг 13.5. Сортировка массива

```
    ;prg_13_5.asm
    MASM
    MODEL   small
    STACK   256
    .data
    mes1 db 0ah,0dh,'Исходный массив - $',0ah,0dh
    ;некоторые сообщения
    mes2 db 0ah,0dh,'Отсортированный массив - $',0ah,0dh
    n    equ 9        ;количество элементов в массиве, считая с 0
mas dw  2,7,4,0,1,9,3,6,5,8  ;исходный массив
tmp dw  0              ;переменные для работы с массивом
i   dw  0
j   dw  0
.code
main:
    mov ax,@data
    mov ds,ax
    xor ax,ax
;вывод на экран исходного массива
    mov ah,09h
    lea dx,mes1
    int 21h            ;вывод сообщения mes1
    mov cx,10
    mov si,0
show_primary:          ;вывод значения элементов
                       ;исходного массива на экран
    mov dx,mas[si]
    add dl,30h
    mov ah,02h
    int 21h
    add si ,2
    loop show_primary

;строки 40-85 программы эквивалентны следующему коду на языке С:
;for (i=0;i<9;i++)
;   for (j=9;j>i;j--)
;       if (mas[i]>mas[j])
;       {tmp=mas[i];
;       mas[i]=mas[j];
;       mas[j]=tmp;}
    mov i,0            ;инициализация i
                       ;внутренний цикл по j
internal:
    mov j,9            ;инициализация j
    jmp cycl_j         ;переход на тело цикла
exchange:
```

```
    mov bx,i          ;bx=i
    shl bx,1
    mov ax,mas[bx]    ;ax=mas[i]
    mov bx,j          ;bx=j
    shl bx,1
    cmp ax,mas[bx]    ;mas[i] ? mas[j] - сравнение элементов
    jle lesser        ;если mas[i] меньше, то обмен не нужен
                      ;и переход на продвижение далее по массиву
                      ;иначе tmp=mas[i], mas[i]=mas[j], mas[j]=tmp:
                      ;tmp=mas[i]
    mov bx,i          ;bx=i
    shl bx,1          ;умножаем на 2, так как элементы - слова
    mov tmp,ax        ;tmp=mas[i]

                      ;mas[i]=mas[j]
    mov bx,j          ;bx=j
    shl bx,1          ;умножаем на 2, так как элементы - слова
    mov ax,mas[bx]    ;ax=mas[j]
    mov bx,i          ;bx=i
    shl bx,1          ;умножаем на 2, так как элементы - слова
    mov mas[bx],ax    ;mas[i]=mas[j]

                      ;mas[j]=tmp
    mov bx,j          ;bx=j
    shl bx,1          ;умножаем на 2, так как элементы - слова
    mov ax,tmp        ;ax=tmp
    mov mas[bx],ax    ;mas[j]=tmp
lesser:               ;продвижение далее по массиву во внутреннем
                      ;цикле
    dec j             ;j--
                      ;тело цикла по j
cycl_j:
    mov ax,j          ;ax=j
    cmp ax,i          ; сравнить j ? i
    jg exchange       ;если j>i, то переход на обмен
                      ;иначе - на внешний цикл по i
    inc i             ;i++
    cmp i,n           ;сравнить i ? n - прошли до конца массива
    jl internal       ;если i<n - продолжение обработки

;вывод отсортированного массива
    mov ah,09h
    lea dx,mes2
    int 21h
prepare:
    mov cx,10
    mov si,0
show:                 ;вывод значения элемента на экран
    mov dx,mas[si]
    add dl,30h
    mov ah,02h
    int 21h
    add si,2
    loop show
exit:
    mov ax,4c00h      ;стандартный выход
    int 21h
end main              ;конец программы
```

В основе программы лежит алгоритм, похожий на метод пузырьковой сортировки. Эта программа не претендует на безусловную оптимальность, так как существует целая теория, касающаяся подобного типа сортировок. Перед нами стоит другая цель — показать использование средств ассемблера для решения подобно-

го рода задач. В программе два цикла. Внешний цикл определяет позицию в массиве очередного элемента, с которым производится попарное сравнение элементов правой части массива (относительно этого элемента). За каждую итерацию внешнего цикла на месте этого очередного элемента оказывается меньший элемент из правой части массива (если он есть). В остальном программа довольно проста и на языке высокого уровня заняла бы около десятка строк. Более подробно с реализацией различных типов сортировок на языке ассемблера можно познакомиться в [8].

Структуры

Рассмотренные нами ранее массивы представляют собой совокупность однотипных элементов. Но часто в приложениях возникает необходимость рассматривать некоторую совокупность данных разного типа как некоторый единый тип. Это очень актуально, например, для программ баз данных, где с одним объектом может ассоциироваться совокупность данных разного типа. К примеру, ранее мы рассмотрели листинг 13.3, в котором работа производилась с массивом трехбайтовых элементов. Каждый элемент, в свою очередь, представлял собой два элемента разных типов: однобайтовое поле счетчика и двухбайтовое поле, которое могло нести еще какую-то нужную для хранения и обработки информацию. Если читатель знаком с одним из языков высокого уровня, то он знает, что такой объект обычно описывается с помощью специального типа данных — структуры. С целью повысить удобство использования языка ассемблера в него также был введен такой тип данных.

По определению *структура* — это тип данных, состоящий из фиксированного числа элементов разного типа.

Для использования структур в программе необходимо выполнить три действия.

1. Задать *шаблон* структуры. По смыслу это означает определение нового типа данных, который впоследствии можно использовать для определения переменных этого типа.
2. Определить *экземпляр* структуры. Этот этап подразумевает инициализацию конкретной переменной с заранее определенной (с помощью шаблона) структурой.
3. Организовать обращение к элементам структуры.

Очень важно хорошо понимать разницу между *описанием* структуры в программе и ее *определением*. Описание структуры в программе означает лишь указание компилятору ее схемы, или шаблона; память при этом не выделяется. Компилятор извлекает из этого шаблона информацию о расположении полей структуры и их значениях по умолчанию. Определение структуры означает указание транслятору на выделение памяти и присвоение этой области памяти символического имени. Описать структуру в программе можно только один раз, а определить — любое количество раз. После того как структура определена, то есть ее имя связано с именем переменной, возможно обращение к полям структуры по их именам.

Описание шаблона структуры

Описание шаблона структуры имеет следующий синтаксис:

```
имя_структуры STRUC
   <описание полей>
имя_структуры ENDS
```

Здесь <описание полей> представляет собой последовательность директив описания данных DB, DW, DD, DQ и DT. Их операнды определяют размер полей и при необходимости — начальные значения. Этими значениями будут, возможно, инициализироваться соответствующие поля при определении структуры.

Как мы уже отметили, при описании шаблона память не выделяется, так как это всего лишь информация для транслятора. Местоположение шаблона в программе может быть произвольным, но, следуя логике работы однопроходного транслятора, шаблон должен быть описан раньше, чем определяется переменная с типом данных структуры. То есть при описании в сегменте данных переменной с типом некоторой структуры ее шаблон необходимо описать в начале сегмента данных либо перед ним.

Рассмотрим работу со структурами на примере моделирования базы данных о сотрудниках некоторого отдела. Для простоты, чтобы уйти от проблем преобразования информации при вводе, условимся, что все поля символьные. Определим структуру записи этой базы данных следующим шаблоном:

```
worker struc                         ;информация о сотруднике
nam         db  30 dup (" ")         ;фамилия, имя, отчество
sex         db  " "                  ;пол
position    db  30 dup (" ")         ;должность
age         db  2 dup (" ")          ;возраст
standing    db  2 dup (" ")          ;стаж
salary      db  4 dup (" ")          ;оклад в рублях
birthdate   db  8 dup (" ")          ;дата рождения
worker  ends
```

Определение данных с типом структуры

Для использования описанной с помощью шаблона структуры в программе необходимо определить переменную с типом данных структуры. Для этого используется следующая синтаксическая конструкция:

```
[имя переменной] имя_структуры <[список значений]>
```

Здесь:

- имя переменной — идентификатор переменной данного структурного типа. Задание имени переменной необязательно. Если его не указать, будет просто выделена область памяти размером в сумму длин всех элементов структуры;
- список значений — заключенный в угловые скобки список начальных значений элементов структуры, разделенных запятыми. Его задание также необязательно. Если список указан не полностью, то все поля структуры для данной переменной инициализируются значениями из шаблона, если таковые заданы. Допускается инициализация отдельных полей, но в этом случае пропущенные поля, которые будут инициализированы значениями из шаблона структуры, должны отделяться запятыми. Если при определении новой переменной с ти-

пом данных структуры мы согласны со всеми значениями полей в ее шаблоне (то есть заданными по умолчанию), то нужно просто написать угловые скобки. К примеру,

victor worker <>.

Для примера определим несколько переменных с типом описанной ранее структуры:

```
data    segment
sotr1   worker  <"Гурко Андрей Вячеславович",,'художник',
'33','15','1800','26.01.64'>
sotr2   worker  <"Михайлова Наталья Геннадьевна",'ж','программист',
'30','10','1680','27.10.58'>
sotr3   worker  <"Степанов Юрий Лонгинович",,'художник',
'38','20','1750','01.01.58'>
sotr4   worker  <"Юрова Елена Александровна",'ж','связист',
'32','2',,'09.01.66'>
sotr5   worker  <>  ;здесь все значения по умолчанию
data    ends
```

Методы работы со структурами

Смысл введения структурного типа данных в любой язык программирования состоит в объединении разнотипных переменных в один объект. В языке должны быть средства доступа к этим переменным внутри конкретного экземпляра структуры. Для того чтобы сослаться в команде на поле некоторой структуры, используется специальный оператор . (точка):

адресное_выражение.имя_поля_структуры

Здесь:

- адресное_выражение — идентификатор переменной некоторого структурного типа или выражение в скобках в соответствии с указанными ранее синтаксическими правилами (рис. 13.1);
- имя_поля_структуры — имя поля из шаблона структуры (это на самом деле тоже адрес, а точнее, смещение поля от начала структуры).

Таким образом, оператор . (точка) вычисляет выражение (адресное_выражение) + + (имя_поля_структуры).

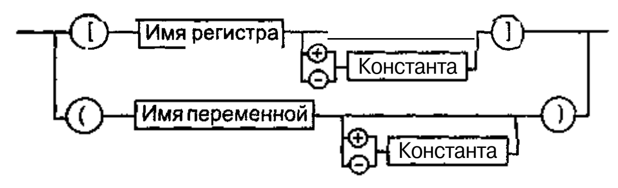


Рис. 13.1. Синтаксис адресного выражения в операторе обращения к полю структуры

Продемонстрируем с помощью определенной нами структуры worker некоторые приемы работы со структурами. К примеру, требуется извлечь в регистр АХ значения поля с возрастом. Так как вряд ли возраст трудоспособного человека может быть больше 99 лет, то после помещения содержимого этого символьного поля в регистр АХ его будет удобно преобразовать в двоичное представление командой AAD (см. главу 8). Будьте внимательны, так как из-за принципа хранения данных «младший байт по младшему адресу» старшая цифра возраста будет поме-

щена в AL, а младшая — в AH. Для корректировки достаточно использовать команду xchg al,ah:

```
    mov ax,word ptr sotr1.age    ;в al возраст sotr1
    xchg ah,al
;а можно и так:
    lea bx,sotr1
    mov ax,word ptr [bx].age
    xchg ah,al
```

Давайте представим, что сотрудников не четверо, а намного больше, и к тому же их число и информация о них постоянно меняются. В этом случае теряется смысл явного определения переменных с типом worker для конкретных личностей. Ассемблер разрешает определять не только отдельную переменную с типом структуры, но и массив структур. К примеру, определим массив из 10 структур типа worker:

```
mas_sotr    worker  10 dup (<>)
```

Дальнейшая работа с массивом структур производится так же, как и с одномерным массивом. Здесь возникает несколько вопросов. Как быть с размером и как организовать индексацию элементов массива?

Аналогично другим идентификаторам, определенным в программе, транслятор назначает имени типа структуры и имени переменной с типом структуры атрибут типа. Значением этого атрибута является размер в байтах, занимаемый полями структуры. Извлечь это значение можно с помощью оператора TYPE. После того как становится известным размер экземпляра структуры, организация индексации в массиве структур не представляет особой сложности. К примеру,

```
worker  struc
...
worker  ends
...
mas_sotr    worker  10 dup (<>)
...
    mov bx,type worker   ;bx=77
    lea di,mas_sotr
;извлечь и вывести на экран пол всех сотрудников:
    mov cx,10
cycl:
    mov dl,[di].sex
... ;вывод на экран содержимого
    ;поля sex структуры worker
    add di,bx       ;к следующей структуре в массиве mas_sort
    loop cycl
```

Как выполнить копирование поля из одной структуры в соответствующее поле другой структуры? Или как выполнить копирование всей структуры? К примеру, выполним копирование поля nam третьего сотрудника в поле nam пятого сотрудника:

```
worker  struc
...
worker  ends
...
mas_sotr    worker  10 dup (<>)
...
    mov bx,offset mas_sotr
    mov si,(type worker)*2   ;si=77*2
    add si,bx
```

```
    mov di,(type worker)*4  ;si=77*4
    add di,bx
    mov cx,30
rep movsb
```

Среди прилагаемых к книге файлов в каталоге ..\lesson13\struct\ приведена программа обслуживания базы данных о сотрудниках[1]. На ее примере вы можете глубже познакомиться с тем, как организовать работу со структурами в своей программе. Возможно, для читателя имеет смысл в полном объеме исследовать эту программу только после знакомства с макрокомандами в следующей главе.

Ремесло программиста рано или поздно делает человека похожим на хорошую домохозяйку. Он, подобно ей, постоянно находится в поиске: где бы чего-нибудь сэкономить, урезать, из минимума продуктов сделать прекрасный обед. И если это удается, то и моральное удовлетворение получается ничуть не меньше, а может, и больше, чем от прекрасного обеда у домохозяйки. Степень этого удовлетворения, как мне кажется, зависит от степени любви к своей профессии. С другой стороны, успехи в разработке программного и аппаратного обеспечения несколько расслабляют программиста, и довольно часто наблюдается ситуация, когда для решения некоторой мелкой задачи привлекаются тяжеловесные средства, эффективность которых в общем случае значима только при реализации сравнительно больших проектов.

В следующих двух разделах описаны два типа данных, наличие которых в языке помогает более эффективно распоряжаться памятью, выделенной программе.

Объединения

Представим ситуацию, когда мы используем некоторую область памяти для размещения того или иного объекта программы (переменной, массива или структуры). Вдруг после некоторого этапа работы у нас отпадает надобность в этих данных. В обычном случае память остается занятой до конца работы программы. Конечно, ее можно было бы задействовать для хранения других переменных, но без принятия специальных мер нельзя изменить тип и имя данных/Неплохо было бы иметь возможность переопределить эту область памяти для объекта с другими типом и именем. Ассемблер предоставляет такую возможность в виде специального типа данных, называемого объединением. *Объединение* — тип данных, позволяющий трактовать одну и ту же область памяти как данные, имеющие разные типы и имена.

Описание объединений в программе напоминает описание структур, то есть сначала указывается шаблон, в котором с помощью директив описания данных перечисляются имена и типы полей:

```
имя_объединения UNION
    <описание полей>
имя_объединения ENDS
```

Отличие объединений от структур состоит, в частности, в том, что при определении переменной типа объединения память выделяется в соответствии с разме-

[1] Все прилагаемые к книге файлы можно найти по адресу http://www.piter.com/download. — *Примеч. ред.*

ром максимального элемента. Обращение к элементам объединения происходит по их именам, но при этом нужно, конечно, помнить, что все поля в объединении накладываются друг на друга. Одновременная работа с элементами объединения исключена. В качестве элементов объединения можно использовать и структуры.

Листинг 13.6, который мы сейчас рассмотрим, примечателен тем, что кроме демонстрации собственно типа данных объединение, в нем показывается возможность взаимного вложения структур и объединений. Постарайтесь внимательно отнестись к анализу этой программы. Основная идея здесь в том, что указатель на память, формируемый программой, может быть представлен в виде:

- 16-разрядного смещения;
- 32-разрядного смещения;
- пары из 16-разрядного смещения и 16-разрядной сегментной составляющей адреса;
- пары из 32-разрядного смещения и 16-разрядного селектора.

Какие из этих указателей можно применять в конкретной ситуации, зависит от режима адресации (use16 или use32) и режима работы процессора. Шаблон объединения, описанный в листинге 13.6, позволяет упростить формирование и использование указателей различных типов.

Листинг 13.6. Пример использования объединения

```
    masm
    model   small
    stack   256
    .586P
    pnt struc          ;структура pnt, содержащая вложенное объединение
        union          ;описание вложенного в структуру объединения
    offs_16 dw  ?
    offs_32 dd  ?
    ends               ;конец описания объединения
segm    dw  ?
    ends               ;конец описания структуры
    .data
    point   union      ;определение объединения,
                       ;содержащего вложенную структуру
    off_16  dw  ?
    off_32  dd  ?
    point_16pnt <>
    point_32pnt <>
    point   ends
    tst db  "Строка для тестирования"
    adr_datapoint   <>  ;определение экземпляра объединения
    .code
main:
    mov ax,@data
    mov ds,ax
    mov ax,seg tst
;записать адрес сегмента строки tst в поле структуры adr_data
    mov adr_data.point_16.segm,ax
;когда понадобится, можно извлечь значение из этого поля обратно,
;к примеру, в регистр bx:
    mov bx,adr_data.point_16.segm
;формируем смещение в поле структуры adr_data
    mov ax,offset tst       ;смещение строки в ax
    mov adr_data.point_16.offs_16,ax
```

продолжение ⇨

Листинг 13.6 *(продолжение)*

```
;аналогично, когда понадобится, можно извлечь
;значение из этого поля:
    mov bx,adr_data.point_16.offs_16
    exit:
    mov ax,4c00h
    int 21h
end main
```

Когда вы будете работать в защищенном режиме процессора и использовать 32-разрядные адреса, то аналогичным способом можете заполнить и задействовать описанное ранее объединение.

Записи

На практике довольно часто возникает необходимость работы с различными программными индикаторами со значениями «включено—выключено». Минимальная величина для операций с памятью — байт. А для программного индикатора достаточно одного бита. То есть из восьми разрядов нужно задействовать всего один. При большом количестве таких индикаторов расход оперативной памяти может быть весьма ощутимым.

Когда мы знакомились с логическими командами, то говорили, что их можно применять для решения подобной проблемы. Но это не совсем эффективно, так как велика вероятность ошибок, особенно при составлении битовых масок. Компиляторы TASM и MASM предоставляют в распоряжение программиста специальный тип данных, использование которого помогает решить проблему работы с битами более эффективно. Речь идет о специальном типе данных — записях. *Запись* — структурный тип данных, состоящий из фиксированного числа элементов длиной от одного до нескольких битов.

При описании записи для каждого элемента указывается его длина в битах и, что необязательно, некоторое значение. Суммарный размер записи определяется суммой размеров ее полей и не может быть более 8, 16 или 32 битов. Если суммарный размер записи меньше указанных значений, то все поля записи «прижимаются» к младшим разрядам.

Использование записей в программе, так же как и структур, организуется в три этапа.

1. Задание шаблона записи, то есть определение набора битовых полей, их длин и, при необходимости, инициализация полей.
2. Определение экземпляра записи. Так же как и для структур, этот этап подразумевает инициализацию конкретной переменной типом заранее определенной с помощью шаблона записи.
3. Организация обращения к элементам записи.

Описание записи

Описание шаблона записи имеет следующий синтаксис:

```
имя_записи RECORD <описание элементов>
```

Здесь <описание элементов> представляет собой последовательность описаний отдельных элементов записи согласно синтаксической диаграмме (рис. 13.2).

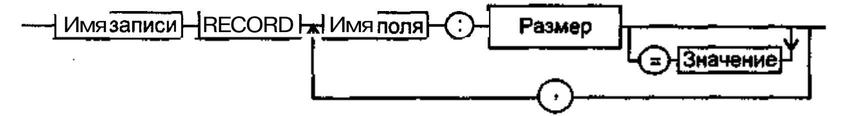


Рис. 13.2. Синтаксис описания шаблона записи

При описании шаблона память не выделяется, так как это всего лишь информация для транслятора ассемблера о структуре записи. Так же как и для структур, местоположение шаблона в программе может быть любым, но при этом необходимо учитывать логику работы однопроходного транслятора.

Определение экземпляра записи

Для использования шаблона записи в программе необходимо определить переменную с типом данной записи, для чего применяется синтаксическая конструкция, показанная на рис. 13.3.

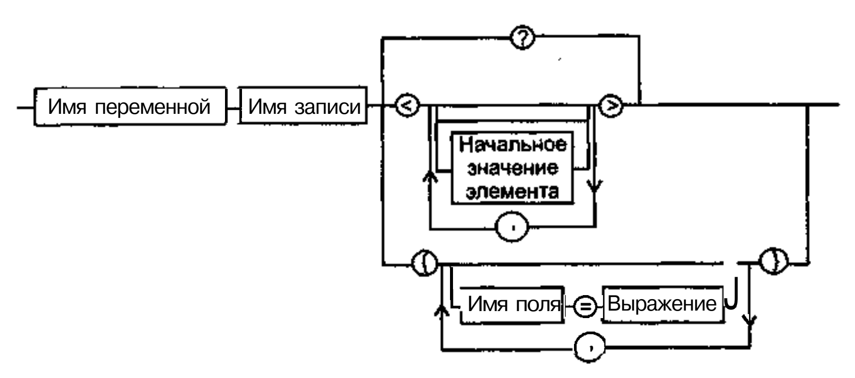


Рис. 13.3. Синтаксис описания экземпляра записи

Анализируя эту синтаксическую диаграмму, можно сделать вывод, что инициализация элементов записи осуществляется довольно гибко. Рассмотрим несколько вариантов инициализации.

Если инициализировать поля не требуется, то дстаточно указать символ ? (вопрос) при определении экземпляра записи:

```
iotest, record  i1:1,i2:2=11,i3:1,i4:2=11,i5:2=00
    ...
flag        iotest      ?
```

Если составить тестовый пример с таким определением записи, то в отладчике можно увидеть, что все поля переменной типа запись flag обнуляются. Это происходит несмотря на то, что в определении записи заданы начальные значения полей.

Если требуется частичная инициализация элементов, то они заключаются в угловые (< и >) или фигурные ({ и }) скобки. Различие здесь в том, что в угловых

скобках элементы должны быть заданы в том же порядке, что и в определении записи. Если значение некоторого элемента совпадает с начальным, то его можно не указывать, но обязательно обозначить запятой. Для последних элементов идущие подряд запятые можно опустить.

К примеру, согласиться со значениями по умолчанию можно так:

```
iotest  record  i1:1,i2:2=11,i3:1,i4:2=11,i5:2=00
...
flag    iotest  <>  ;согласились со значением по умолчанию
```

Изменить значение поля i2 можно так:

```
iotest  record  i1:1,i2:2=11,i3:1,i4:2=11,i5:2=00
...
flag    iotest  <,10,>  ; переопределили i2
```

Применяя фигурные скобки, можно провести также выборочную инициализацию полей, но при этом обозначать запятыми поля, со значениями по умолчанию которых мы согласны, не обязательно:

```
iotest  record  i1:1,i2:2=11,i3:1,i4:2=11,i5:2=00
...
flag    iotest  {i2=10} ;переопределили i2, не обращая
                        ;внимания на порядок
                        ;следования других компонентов записи
```

Работа с записями

Как организовать работу с отдельными элементами записи? Обычные механизмы адресации здесь бессильны, так как они работают на уровне ячеек памяти, то есть байтов, а не отдельных битов. Здесь программисту нужно приложить некоторые усилия. Прежде всего, для понимания проблемы нужно усвоить несколько моментов.

- Каждому имени элемента записи ассемблер присваивает числовое значение, равное количеству сдвигов вправо, которые нужно произвести для того, чтобы этот элемент оказался «прижатым» к началу ячейки. Это дает нам возможность локализовать его и работать с ним. Но для этого нужно знать длину элемента в битах.
- Сдвиг вправо производится с помощью команды сдвига SHR.
- Ассемблер содержит оператор WIDTH, который позволяет узнать размер элемента записи в битах или полностью размер записи. Варианты применения оператора WIDTH:
  - width имя_элемента_записи — значением оператора будет размер элемента в битах;
  - width имя_экземпляра_записи или width имя_типа_записи — значением оператора будет размер всей записи в битах.

  Например,

  ```
  mov al,width i2
  ...
  mov ax,width iotest
  ```
- Ассемблер содержит оператор MASK, который позволяет локализовать биты нужного элемента записи. Эта локализация производится путем создания маски,

размер которой совпадает с размером записи. В этой маске должны быть обнулены биты во всех позициях, за исключением тех, которые занимает элемент в записи.

- Сами действия по преобразованию элементов записи производятся с помощью логических команд.

Подчеркнем еще раз то обстоятельство, что непосредственное обращение к конкретному элементу записи невозможно. Для этого нужно сначала выделить его, сдвинуть при необходимости к младшим разрядам, выполнить требуемые действия и поместить обратно на свое место в записи. Поэтому, чтобы вам каждый раз «не изобретать велосипед», далее мы опишем типовые алгоритмы осуществления этих операций над элементами записи. Вам останется лишь оформить эти алгоритмы в виде кода соответствии *с* требованиями конкретной задачи.

Для выделения элемента записи требуется выполнить описанную далее процедуру.

1. Поместить запись во временную память — регистр (8-, 16- или 32-разрядный в зависимости от размера записи).
2. Получить битовую маску, соответствующую элементу записи, с помощью оператора MASK.
3. Локализовать биты в регистре с помощью маски и команды AND.
4. Сдвинуть биты элемента к младшим разрядам регистра командой SHR. Число разрядов для сдвига получить с использованием имени элемента записи.

В результате этих действий элемент записи будет локализован в начале рабочего регистра, и далее с ним можно производить любые действия (см. далее).

В ходе предыдущих рассуждений мы показали, что с элементами записи производятся любые действия, как с обычной двоичной информацией. Единственное, что нужно отслеживать, — это размер битового поля. Если, к примеру, размер поля увеличится, то впоследствии может произойти случайное изменение соседних полей битов. Поэтому желательно на этапе проектирования предусматривать все варианты функционирования программы с тем, чтобы исключить любые изменения размеров полей.

Измененный элемент помещается на его место в запись следующим образом.

1. Используя имя элемента записи в качестве счетчика сдвигов, сдвинуть влево биты элемента записи.
2. Если вы не уверены в том, что разрядность результата преобразований не превысила исходную, можно выполнить «обрезание» лишних битов, используя команду AND и маску элемента.
3. Подготовить исходную запись к вставке измененного элемента путем обнуления битов в записи на месте этого элемента. Это можно сделать наложением командой AND инвертированной маски элемента записи на исходную запись.
4. С помощью команды OR наложить значение в регистре на исходную запись.

В качестве примера рассмотрим листинг 13.7, в котором поле i2 в записи iotest обнуляется.

10 Зак. 256

Листинг 13.7. Работа с полем записи

```
;prg_13_7.asm
masm
model    small
stack   256
iotest   record   i1:1,i2:2=11,i3:1,i4:2=11,i5:2=00
.data
flag    iotest  <>
.code
main:
    mov ax,@data
    mov ds,ax
    mov al,mask i2
    shr al,i2        ;биты i2 в начале ax
    and al,0fch      ;обнулили i2
;помещаем i2 на место
    shl al,i2
    mov bl,[flag]
    xor bl,mask i2   ;сбросили i2
    or  bl,al        ;наложили
    exit:
    mov ax,4c00h;стандартный выход
    int 21h
end main             ;конец программы
```

В заключение стоит еще раз проанализировать приведенные сведения о записях и особенностях работы с ними. При этом обратите внимание на то обстоятельство, что мы нигде явно не просчитываем расположение битов. Поэтому, если понадобится изменить размер элемента или его начальное значение, то достаточно лишь внести изменения в экземпляр записи или в описание ее типа. Функциональную часть программы, работающую с этой записью, трогать не нужно.

Дополнительные возможности обработки

Понимая важность типа данных «запись» для эффективного программирования, разработчики транслятора TASM, начиная с версии 3.0, включили в систему его команд две дополнительные команды на правах директив. Последнее означает, что эти команды внешне имеют формат обычных команд ассемблера, но после трансляции они приводятся к одной или нескольким машинным командам. Введение этих команд в язык TASM повышает наглядность работы с записями, оптимизирует код и уменьшает размер программы. Эти команды позволяют скрыть от программиста действия по выделению и установке отдельных полей записи (см. ранее).

Для установки значения некоторого поля записи используется команда SETFIELD с синтаксисом:

```
setfield имя_элемента_записи приемник,регистр_источник
```

Для выборки значения некоторого поля записи используется команда GETFIELD с синтаксисом:

```
getfield имя_элемента_записи регистр_приемник,источник
```

Работа команды SETFIELD заключается в следующем. Местоположение записи определяется операндом приемник, который может представлять собой имя регистра или адрес памяти. Операнд имя_элемента_записи определяет элемент записи, с которым ведется работа (по сути, если вы были внимательны, он определяет смещение элемента в записи относительно младшего разряда). Новое значение, в ко-

торое необходимо установить указанный элемент записи, должно содержаться в операнде регистр_источник. Обрабатывая данную команду, транслятор генерирует последовательность команд, которые выполняют следующие действия.

1. Сдвиг содержимого операнда регистр_источник влево на количество разрядов, соответствующее расположению элемента в записи.
2. Выполнение логической операции OR над операндами приемник и регистр_источник. Результат операции помещается в операнд приемник.

Важно отметить, что SETFIELD не производит предварительной очистки элемента, в результате после логического сложения командой OR возможно наложение старого содержимого элемента и нового устанавливаемого значения. Поэтому требуется предварительно подготовить поле в записи путем его обнуления.

Действие команды GETFIELD обратно действию SETFIELD. В качестве операнда источник может быть указан либо регистр, либо адрес памяти. В регистр, указанный операндом регистр_приемник, помещается результат работы команды — значение элемента записи. Интересная особенность связана с операндом регистр_приемник. Команда GETFIELD всегда использует 16-разрядный регистр, даже если вы укажете в этой команде имя 8-разрядного регистра.

В качестве примера применения команд SETFIELD и GETFIELD рассмотрим листинг 13.8.

Листинг 13.8. Работа с полями записи

```
;prg_13_8.asm
masm
model   small
stack   256
iotest  record  i1:1,i2:2=11,i3:1,i4:2=11,i5:2=00
.data
flag    iotest  <>
.code
main:
    mov ax,@data
    mov ds,ax
    mov al,flag
    mov bl,3
    setfield i5 al,bl
    xor bl,bl
    getfield i5 bl,al
    mov bl,1
    setfield i4 al,bl
    setfield i5 al,bl
exit:
    mov ax,4c00h        ;стандартный выход
    int 21h
end main                ;конец программы
```

В листинге 13.8 демонстрируется порядок извлечения и установки некоторых полей записи. Результат работы команд SETFIELD и GETFIELD удобнее всего изучать в отладчике. При установке значений полей не производится их предварительная очистка. Это сделано специально. Для такого рода операций лучше использовать некоторые универсальные механизмы, иначе велик риск внесения ошибок, которые трудно обнаружить и исправить. В качестве такого механизма можно предложить макрокоманды, к рассмотрению которых мы и приступим в следующей главе.

В заключение хотелось бы привести еще один пример использования записей. Это описание регистра EFLAGS. Для удобства мы разбили описание на три части:

- eflags_l_7 — младший байт регистра EFLAGS/FLAGS;
- eflags_8_15 — второй байт регистра EFLAGS/FLAGS;
- eflags_h — старшая половина регистра EFLAGS.

```
eflags_l_7   record
sf7:1=0,zf6:1=0,c5:1=0,af4:1=0,c3:1=0,pf2:1=0,c1:=1,cf0:1=0
eflags_l_15  record
c15:1=0,nt14:1=0,iopl:2=0,of11:1=0,df10:1=0,if9:1=1,tf8:1=0
eflags_h     record  c:13=0,ac18:1=0,vm17:1=0,rf16:1=0
```

К этому описанию стоит вернуться после изучения материала следующей главы, посвященного макрокомандам. Возможно, что для работы с регистром флагов вам захочется разработать соответствующую макрокоманду. Подобная хорошо протестированная макрокоманда в будущем поможет вам избежать многих трудно обнаруживаемых ошибок.

Итоги

- TASM поддерживает несколько дополнительных типов данных, значительно расширяющих возможности базовых директив резервирования и инициализации данных. По сути, эти типы заимствованы из языков высокого уровня и призваны облегчить разработку прикладных программ на ассемблере.
- Практическое использование дополнительных типов данных требует повышенного внимания и отражает специфику программирования на языке ассемблера.
- Понятия массива и индексации массива весьма условны, и логическая интерпретация области памяти, отведенной под массив, определяется алгоритмом обработки.
- Тип структуры в языке ассемблера позволяет создать совокупность логически взаимосвязанных разнотипных данных и рассматривать их как отдельный объект. Это очень удобно, когда в программе необходимо иметь несколько таких объектов. В этом случае обычно организуют массив структур.
- Основное достоинство объединений — в возможности «плюрализма суждений» о типе одной и той же области памяти.
- Записи в языке ассемблера расширяют возможности логических команд для работы на уровне битов, что подчеркивает значение ассемблера как языка системного программирования.

Глава 14
Макросредства языка ассемблера

- Понятие о макросредствах языка ассемблера
- Псевдооператоры EQU и =
- Макрокоманды и макродирективы
- Директивы условной компиляции
- Директивы генерации ошибок пользователя
- Директивы управления листингом

Язык ассемблера не относится к простым языкам программирования. По истечении некоторого времени после начала программирования на ассемблере становятся видны свойственные этому языку проблемы. Перечислим некоторые из них:

- плохая читаемость исходных текстов программы — спустя некоторое время при недостаточном комментировании могут возникнуть проблемы с пониманием особенностей алгоритма, лежащего в основе программы;
- ограниченность набора команд;
- полная или почти полная повторяемость некоторых фрагментов программы;
- необходимость включения в каждую новую программу одних и тех же фрагментов кода и т. д.

При написании программы на машинном языке данные проблемы были бы принципиально неразрешимы. Но язык ассемблера, являясь символическим представлением машинного языка, в то же время предоставляет ряд средств более высокого уровня для их решения. Основной целью, которая при этом преследуется,

является повышение удобства написания программ. В общем случае эта цель достигается за счет:

- расширения набора директив;
- введения некоторых дополнительных команд, не имеющих аналогов в системе команд процессора (за примером далеко ходить не нужно — это рассмотренные в главе 13 команды SETFIELD и GETFIELD, которые скрывают от программиста рутинные действия и генерируют наиболее эффективный код);
- введения сложных типов данных.

Однако этот перечень исчерпывает все глобальные направления, по которым развивается транслятор от версии к версии. Что же делать программисту для решения его локальной задачи, для адаптации процесса разработки к нуждам определенной проблемной области? Для этого разработчики компиляторов ассемблера включают в язык и постоянно совершенствуют аппарат *макросредств.* Этот аппарат является очень мощным и важным. В общем случае есть смысл говорить о том, что транслятор ассемблера состоит из двух частей — непосредственно транслятора, формирующего объектный модуль, и *макроассемблера* (рис. 14.1). Если вы знакомы с языком С или С++, то, конечно, знаете используемый в нем механизм препроцессорной обработки. Основные принципы его работы аналогичны принципам работы макроассемблера. Для тех, кто ничего раньше не слышал об этом механизме, поясню его суть. Основная идея — использование подстановок, когда определенным образом организованная символьная последовательность заменяется другой символьной последовательностью. Создаваемая последовательность может описывать как данные, так и программные коды. Главное здесь то, что на входе макроассемблера текст программы может быть весьма далеким по виду от ассемблера, но на выходе обязательно будет текст на чистом ассемблере, содержащем символические аналоги машинных команд процессора. Таким образом, обработка программы на ассемблере с использованием макросредств неявно осуществляется транслятором в две фазы. На первой фазе работает часть компилятора,

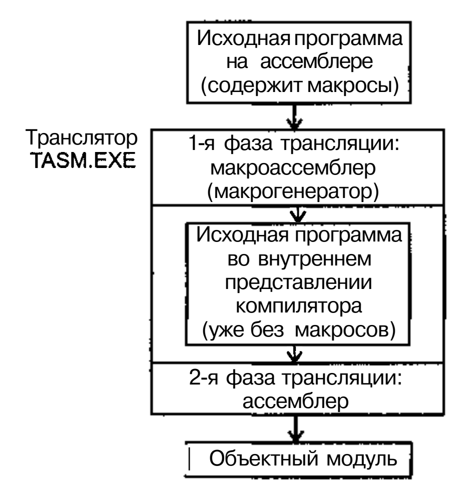


Рис. 14.1. Макроассемблер в общей схеме трансляции программы для компилятора TASM

называемая макроассемблером, основные функции которого мы описали ранее. Во второй фазе трансляции участвует непосредственно ассемблер, задачей которого является формирование объектного кода, содержащего текст исходной программы в машинном виде.

Далее мы обсудим основной набор макросредств компилятора TASM. Отметим, что большинство этих средств доступно и в компиляторе MASM фирмы Microsoft. Обсуждение начнем с простейших средств и закончим более сложными.

Псевдооператоры EQU и =

К простейшим макросредствам языка ассемблера можно отнести псевдооператоры EQU и = (равно). Их мы уже неоднократно использовали при написании программ. Данные псевдооператоры предназначены для присвоения некоторому выражению символического имени или идентификатора. Впоследствии, когда в ходе трансляции такие идентификаторы встречаются в теле программы, макроассемблер подставляет вместо них соответствующее выражение. Выражениями могут быть константы, имена меток, символические имена и строки в апострофах. После присвоения этим конструкциям символических имен их можно использовать везде, где эти конструкции могли бы присутствовать.

Синтаксис псевдооператора EQU:

```
имя_идентификатора equ строка или числовое_выражение
```

Синтаксис псевдооператора =:

```
имя_идентификатора = числовое_выражение
```

Несмотря на внешнее и функциональное сходство, псевдооператоры EQU и = различаются следующим:

- с помощью псевдооператора EQU идентификатору можно ставить в соответствие как числовые выражения, так и текстовые строки, а псевдооператор = может использоваться только с числовыми выражениями;
- идентификаторы, определенные с помощью псевдооператора =, можно переопределять в исходном тексте программы, а определенные с использованием псевдооператора EQU — нельзя.

Ассемблер всегда пытается вычислить значение *строки,* воспринимая ее как выражение. Для того чтобы строка воспринималась именно как текстовая, необходимо заключить ее в угловые скобки:

```
<строка>
```

Угловые скобки являются оператором ассемблера, называемым оператором *выделения. С* его помощью транслятору сообщается, что заключенная в угловые скобки строка должна трактоваться как текст, даже если в нее входят служебные слова ассемблера или операторы.

Псевдооператор EQU удобно использовать для настройки программы на конкретные условия выполнения, замены сложных в обозначении объектов, многократно встречающихся в программе, более простыми именами и т. п. К примеру,

```
masm
model    small
stack   256
```

```
mas_size    equ 10          ;размерность массива
akk equ ax                  ;переименовать регистр
mas_elem    equ mas[bx][si] ;адресовать элемент массива
.data
;описание массива из 10 байт:
mas db  mas_size dup  (0)
.code
    mov akk,@data           ;фактически mov ax,@data
    mov ds,akk              ;фактически mov ds,ax
...
    mov al,mas_elem         ;фактически mov al,mas[bx][si]
```

Псевдооператор = удобно использовать для определения простых абсолютных (то есть не зависящих от места загрузки программы в память) математических выражений. Главное условие — транслятор должен быть в состоянии вычислять эти выражения во время трансляции. К примеру,

```
.data
adr1    db  5 dup  (0)
adr2    dw  0
len = 43
len = len+1                 ;можно и так, через предыдущее определение
len = adr2-adr1
```

Как видно из примера, в правой части псевдооператора = можно использовать метки и ссылки на адреса — главное, чтобы в итоге получилось абсолютное выражение.

Компилятор TASM, начиная с версии 3.00, содержит директивы, значительно расширяющие его возможности по работе с текстовыми макросами. Эти директивы аналогичны некоторым функциям обработки строк в языках высокого уровня. Под *строками* здесь понимается текст, описанный с помощью псевдооператора EQU. Эти директивы перечислены далее.

- Директива слияния строк CATSTR:

  ```
  идентификатор catstr строка_1,строка_2,...
  ```

  Значением этого макроса будет новая строка, состоящая из сцепленной слева направо последовательности строк строка_1, строка_2 и т. д. В качестве сцепляемых строк могут быть указаны имена ранее определенных макросов. К примеру,

  ```
  pre equ Привет,
  name    equ < Юля>
  privet  catstr  pre,name            ;privet= "Привет, Юля"
  ```

- Директива выделения подстроки в строке SUBSTR:

  ```
  идентификатор  substr  строка,номер_позиции,размер
  ```

  Значением данного макроса будет часть строки, заданной операндом строка, начинающаяся с позиции с номером номер_позиции и длиной, указанной операндом размер. Если требуется только остаток строки начиная с некоторой позиции, то достаточно указать номер позиции (без размера). К примеру,

  ```
  ;продолжение предыдущего фрагмента:
  privet  catstr      pre,name        ;privet= "Привет, Юля"
  name    substr      privet,7,3      ;name="Юля"
  ```

- Директива определения вхождения одной строки в другую INSTR:

  ```
  идентификатор instr номер_нач_позиции,строка_1,строка_2
  ```

  После обработки данного макроса транслятором операнду идентификатор будет присвоено числовое значение, соответствующее номеру (первой) позиции,

с которой совпадают операнды строка_1 и строка_2. Если такого совпадения нет, то идентификатор получит значение 0.

- Директива определения длины строки в текстовом макросе SIZESTR:

  идентификатор sizestr строка

  В результате обработки данного макроса значение идентификатор устанавливается равным длине строки:

```
;как продолжение предыдущего фрагмента:
privet   catstr     pre,name   ;privet= "Привет, Юля"
len      sizestr privet         ;len=10
```

Эти директивы очень удобно использовать при разработке *макрокоманд,* которые являются следующим типом макросредств, предоставляемых компилятором ассемблера.

Макрокоманды

По смыслу макрокоманда представляет собой дальнейшее развитие механизма замены текста. С помощью макрокоманд в текст программы можно вставлять последовательности строк (которые логически могут быть данными или командами) и даже более того — привязывать их к контексту места вставки.

Представим ситуацию, когда необходимо выполнить некоторые повторяющиеся действия. Механизм макрокоманд предоставляет возможность заменить одинаковые участки кода одной строкой, а первоначальный исходный текст описать один раз в определенном месте программы или во внешнем файле. Дальнейшее наше обсуждение будет посвящено тому, как это сделать.

Определимся с терминологией. Осмысленное рассмотрение данного механизма предполагает понимание смысла терминов *макрокоманда* и *макроопределение. Макрокоманда* представляет собой строку, одним из компонентов которой является символическое *имя макрокоманды.* Имя макрокоманды может сопровождаться параметрами. Если данная строка встречается в теле исходного текста программы, то транслятор замещает ее одной или несколькими другими строками. Какими именно строками — определяется *макроопределением,* которое представляет собой *шаблон* (описание) макрокоманды.

Таким образом, для использования макрокоманды в программе первым делом задают ее *макроопределение.* Синтаксис макроопределения следующий:

```
имя_макрокоманды macro список_формальных_аргументов
тело макроопределения
endm
```

Где должны располагаться макроопределения? Есть три варианта.

- Макроопределения могут располагаться в начале исходного текста программы до сегмента кода и данных с тем, чтобы не ухудшать читабельность программы. Этот вариант следует применять в случаях, если определяемые вами макрокоманды актуальны только в пределах одной этой программы.
- Макроопределения могут располагаться в отдельном файле. Этот вариант подходит при работе над несколькими программами одной проблемной области. Чтобы сделать доступными эти макроопределения в конкретной программе,

необходимо в начале исходного текста этой программы записать директиву include имя_файла, к примеру:

```
masm
model       small
include show.inc
;в это место будет вставлен текст файла show.inc
...
```

- Макроопределения могут располагаться в макробиблиотеке. Если у вас есть универсальные макрокоманды, которые используются практически во всех ваших программах, то их целесообразно записать в так называемую *макробиблиотеку.* Сделать актуальными макрокоманды из этой библиотеки можно с помощью все той же директивы INCLUDE. Недостаток этого и предыдущего способов — в том, что в исходный текст программы включаются абсолютно все макроопределения. Для исправления ситуации можно задействовать директиву PURGE, в качестве операндов которой через запятую перечисляются имена макрокоманд, которые не должны включаться в текст программы. К примеру,

```
...
include iomac.inc
purge       outstr,exit
...
```

В данном случае в исходный текст программы перед началом компиляции транслятор TASM вместо строки include iomac.inc вставит строки из файла iomac.inc, отличающиеся от оригинала тем, что в этих строках будут отсутствовать макроопределения OUTSTR и EXIT.

А теперь вернемся к главе 6 и вспомним программу из листинга 6.1. Проанализируем ее текст, выявим повторяющиеся участки и составим для них макроопределения (листинг 14.1).

Листинг 14.1. Первый пример создания и использования макрокоманд

```
;prg_6_1.asm с макроопределениями
init_ds macro
;Макрос настройки ds на сегмент данных
    mov ax,data
    mov ds,ax
    endm
out_str macro   str
;Макрос вывода строки на экран.
;На входе - выводимая строка.
;На выходе - сообщение на экране.
        push ax
        mov ah,09h
        mov dx,offset str
        int 21h
        pop ax
        endm

clear_r macro   rg
;очистка регистра rg
    xor rg,rg
    endm

get_char macro
;ввод символа
;введенный символ в al
```

```
<26>          mov ah,1h
<27>          int 21h
<28>          endm
<29>
<30>      conv_16_2   macro
<31>      ;макрос преобразования символа шестнадцатеричной цифры
<32>      ;в ее двоичный эквивалент в al
<33>          sub al,30h
<34>          cmp al,9h
<35>          jle $+4
<36>          sub al,7h
<37>          endm
<38>
<39>      exit macro
<40>      ;макрос конца программы
<41>          mov ax,4c00h
<42>          int 21h
<43>          endm
<44>
<45>      datasegment para public "data"
<46>      message db  "Введите две шестнадцатеричные цифры (буквы
A,B,C,D,E,F - прописные): $"
<47>      dataends
<48>      stk segment stack
<49>          db  256 dup("?")
<50>      stk ends
<51>
<52>      codesegment para public "code"
<53>          assume  cs:code,ds:data,ss:stk
<54>      main   proc
<55>          init_ds
<56>          out_str message
<57>          clear_r ax
<58>          get_char
<59>          conv_16_2
<60>          mov dl,al
<61>          mov cl,4h
<62>          shl dl,cl
<63>          get_char
<64>          conv_16_2
<65>          add dl,al
<66>          xchgdl,al   ;результат в al
<67>          exit
<68>      main endp
<69>      code ends
<70>      end main
```

В листинге 14.1 в строках 2-7, 8–16, 18–21, 23–28, 30-37, 39-43 описаны макроопределения. Описание их назначения приведено сразу после заголовка в теле каждого макроопределения. Если вынести эти макроопределения в отдельный файл, то впоследствии можно использовать их при написании других программ. Посмотрите на модернизированный исходный текст программы из листинга 6.1 в листинге 14.1 (строки 54-68). Если не обращать внимания на некоторые неясные моменты, то сам сегмент кода стал внешне более читабельным, и даже можно сказать, что в его новых командах появился какой-то смысл. Откомпилируйте листинг 14.1 и получите файл листинга. После этого сравните текст программы до и после обработки ассемблером. Вы увидите, что исходный текст изменился, — после строк программы, в которых были макрокоманды, появились фрагменты текста. Содержимое этих фрагментов определяется содержимым макроопределений (шаблонов), заданных в начале программы. Видно, что путем ввода параметров

для макрокоманды можно модернизировать содержимое фактического текста, замещающего макрокоманду.

Функционально макроопределения похожи на процедуры. Сходство в том, что и те, и другие достаточно один раз где-то описать, а затем вызывать их специальным образом. На этом сходство заканчивается и начинаются различия, которые в зависимости от целевой установки можно рассматривать и как достоинства, и как недостатки.

- В отличие от процедуры, текст которой неизменен, макроопределение в процессе макрогенерации может меняться в соответствии с набором фактических параметров. При этом коррекции могут подвергаться как операнды команд, так и сами команды. Процедуры в этом отношении менее гибки.
- При каждом вызове макрокоманды ее текст в виде макрорасширения вставляется в программу. При вызове процедуры процессор осуществляет передачу управления на начало процедуры, находящейся в некоторой области памяти в одном экземпляре. Код в этом случае получается более компактным, хотя быстродействие несколько снижается за счет необходимости осуществления переходов.

Макроопределение обрабатывается компилятором особым образом. Для того чтобы использовать описанное макроопределение, его нужно «активизировать» с помощью *макрокоманды*. Для этого в нужном месте исходного кода программы на основе текста заголовка макроопределения указывается следующая синтаксическая конструкция:

```
имя_макрокоманды список_фактических_аргументов
```

В результате применения данной синтаксической конструкции соответствующая строка исходного текста программы заменяется строками из тела макроопределения. Но это не простая замена. Обычно макрокоманда содержит некоторый списокаргументов(список_фактических_аргументов),которымикорректируется макроопределение. Места в теле макроопределения, которые будут замещаться фактическими аргументами из макрокоманды, обозначаются с помощью так называемых *формальных* аргументов. Таким образом, в результате применения макрокоманды в программе формальные аргументы в макроопределении замещаются соответствующими фактическими аргументами; в этом и заключается учет контекста. Процесс такого замещения называется *макрогенерацией,* а результатом этого процесса является *макрорасширение.*

К примеру, рассмотрим самое короткое макроопределение в листинге 14.1 — clear_rg. Как отмечено ранее, результаты работы макроассемблера можно узнать, просмотрев файл листинга после трансляции. Покажем несколько его фрагментов, которые демонстрируют, как был описан текст макроопределения clear_rg (строки 24-27), как был осуществлен вызов макрокоманды clear_rg с фактическим параметром ах (строка 74) и как выглядит результат работы макрогенератора, сформировавшего команду ассемблера xor ax,ax (строка 75):

```
<24>        clear_r macro   rg
<25>         ;очистка регистра rg
<26>        xor rg,rg
<27>        endm
...
```

```
<74>        clear_r ax
<75>    000E 33 C0  xor ax,ax
```

В итоге мы получили то, что и требовалось, — команду очистки заданного регистра, в данном случае AX. В другом месте программы вы можете выдать ту же макрокоманду, но уже с другим именем регистра. Если у вас есть желание, можете провести эксперименты с этой и другими макрокомандами. Каждый фактический аргумент макрокоманды представляет собой строку символов, для формирования которой применяются определенные правила.

- Строка может состоять:
  - из последовательности символов без пробелов, точек, запятых, точек с запятой;
  - из последовательности любых символов, заключенных в угловые скобки (<...>), в которой можно указывать как пробелы, так и точки, запятые, точки с запятыми (мы упоминали об этом операторе выделения ассемблера при обсуждении директивы EQU).
- Для того чтобы указать, что некоторый символ внутри строки, представляющей фактический параметр, является собственно символом, а не чем-то иным, например, некоторым разделителем или ограничивающей скобкой, применяется специальный оператор !. Этот оператор ставится непосредственно перед нужным символом, и его действие эквивалентно заключению этого символа в угловые скобки. Таким образом, оператор ! также является оператором выделения, но одиночного символа.
- Для вычисления в строке, представляющей фактический параметр в макрокоманде, некоторого константного выражения в начале этого выражения нужно поставить знак % (процент):

  `% константное_выражение`

  Значение **константное_выражение** вычисляется и подставляется в текстовом виде в соответствии с текущей системой счисления[1].

Теперь обсудим вопрос, как транслятор распознает формальные аргументы в теле макроопределения для их последующей замены фактическими аргументами?

Прежде всего по их именам в заголовке макроопределения. В процессе генерации макрорасширения компилятор ассемблера ищет в тексте тела макроопределения последовательности символов, совпадающие с теми последовательностями символов, из которых состоят формальные параметры. После обнаружения такого совпадения формальный параметр из тела макроопределения замещается соответствующим фактическим параметром из макрокоманды. Этот процесс называется *подстановкой аргументов*. Здесь нужно отметить еще раз особо список формальных аргументов в заголовке макроопределения. В общем случае он может содер-

[1] Под текущей системой счисления понимается то, как интерпретируются транслятором числа или строки с фиксированным числовым значением — как двоичные, десятичные или шестнадцатеричные числа. По умолчанию транслятор трактует их как десятичные. Ассемблер имеет специальную директиву **.radix**, которая дает возможность изменить текущую систему счисления. В качестве операнда директивы **.radix** может быть значение **2, 10** или 16, что означает выбор, соответственно, двоичной, десятичной или шестнадцатеричной системы счисления.

жать не только разделенные запятыми формальные элементы, но и некоторую дополнительную информацию. Полный синтаксис формального аргумента следующий:

```
имя_формального_аргумента[:тип]
```

Здесь тип может принимать значения:

- REQ — это значение говорит о том, что требуется обязательное явное задание фактического аргумента при вызове макрокоманды;
- m =<любая_строка> — если аргумент при вызове макрокоманды не задан, то в соответствующие места в макрорасширении по умолчанию будет вставлено значение любая_строка (будьте внимательны: символы, входящие в операнд любая_строка, должны быть заключены в угловые скобки).

Однако распознать в теле макроопределения формальный аргумент ассемблер может не всегда. Это, например, может не произойти в случае, когда он является частью некоторого идентификатора. В этом случае последовательность символов формального аргумента отделяют от остального контекста с помощью специального оператора *замены* (символа &). Этот прием часто используется для задания модифицируемых идентификаторов и кодов операций. К примеру, определим макрос, который предназначен для генерации в программе некоторой таблицы, причем параметры этой таблицы можно задавать с помощью аргументов макрокоманды:

```
...
def_table   macro   type:=b,len:REQ
tabl_&type  d&type  len dup (0)
endm
...
.data
def_tabl    b,10
def_tabl    w,5
```

После трансляции текста программы, содержащей эти строки, получатся следующие макрорасширения:

```
tabl_b  db  10 dup (0)
tabl_w  dw  10 dup (0)
```

Символ & можно применять и для распознавания формального аргумента в строке, заключенной в двойные кавычки (" "). Например,

```
num_char    macro   message
;...
;подсчитать количество (num) символов в строке
    jmp ml
elem    db  "Строка &message содержит"
;число символов в строке message в коде ASCII
num db  2 dup (0)
    db  " символов",10,13,'$'   ;конец строки
                                ;для вывода функцией 09h
m1:
;...
;вывести elem на экран
    endm
```

В связи с последним фрагментом разберем ситуацию, когда тело макроопределения содержит метку или имя в директиве резервирования и инициализации данных. Если некоторая макрокоманда вызывается в программе несколько раз, то при

макрогенерации один и тот же идентификатор будет определен несколько раз, что, естественно, транслятор посчитает ошибкой. Для выхода из подобной ситуации применяют директиву LOCAL, которая имеет следующий синтаксис:

```
local список_идентификаторов
```

Эту директиву необходимо указывать непосредственно за заголовком макроопределения. Результатом работы директивы LOCAL будет генерация в каждом экземпляре макрорасширения уникальных имен для всех идентификаторов, перечисленных в операнде **список_идентификаторов**. Эти уникальные имена имеют вид ??xxxx, где xxxx — шестнадцатеричное число. Для первого идентификатора в первом экземпляре макрорасширения xxxx = 0000, для второго — xxxx = 0001 и т. д. Контроль правильности размещения и использования этих уникальных имен берет на себя транслятор ассемблера. Для того чтобы окончательно разобраться в деталях, введем и оттранслируем листинг 14.2. В нем помимо некоторых ранее рассмотренных макрокоманд содержится макрокоманда num_char. Ее назначение — подсчитывать количество символов в строке, адрес которой передается этой макрокоманде в качестве фактического параметра. Строка должна удовлетворять требованию, предъявляемому к строке, предназначенной для вывода на экран функцией 09h прерывания 21h, то есть заканчиваться символом $. Другой момент, который нашел отражение в этой программе, — использование символа & для распознавания формального аргумента в строке, заключенной в кавычки " " (см. последний фрагмент).

Листинг 14.2. Второй пример создания и использования макрокоманд

```
;prg_14_2.asm
init_ds macro
;макрос настройки ds на сегмент данных
    mov ax,data
    mov ds,ax
    xor ax,ax
    endm
out_str macro   str
;макрос вывода строки на экран.
;На входе - выводимая строка.
;На выходе - сообщение на экране.
    push ax
    mov ah,09h
    mov dx,offset str
    int 21h
    pop ax
    endm
exit    macro
;макрос конца программы
    mov ax,4c00h
    int 21h
    endm
num_char    macro   message
    local   m1,elem,num,err_mes,find,num_exit
;макрос подсчета количества символов в строке.
;Длина строки - не более 99 символов.
;Вход: message - адрес строки символов, ограниченной "$"
;Выход: в al - количество символов в строке message
;и вывод сообщения
    jmp ml
elem    db  "Строка &message содержит "
```

продолжение ⇗

Листинг 14.2 *(продолжение)*

```
num db  2 dup (0)              ;число символов в строке
                               ;message в коде ASCII
    db  " символов",10,13,'$'  ;конец строки
                               ;для вывода функцией 09h
err_mes db  "Строка &message не содержит символа конца строки",10,13,'$'
m1:
;сохраняем используемые в макросе регистры
    push es
    push cx
    push ax
    push di
    push ds
    pop es                     ;настройка es на ds
    mov al,'$'                 ;символ для поиска - "$"
    cld                        ;сброс флага df
    lea di,message             ;загрузка в es:di смещения
                               ;строки message
    pushdi                     ;запомним di - адрес начала строки
    mov cx,99                  ;для префикса repne - максимальная
                               ;длина строки
                               ;поиск в строке (пока нужный символ
                               ;и символ в строке не равны)
                               ;выход - при первом совпавшем
repne   scasb
    je  find                   ;если символ найден - переход на обработку
;вывод сообщения о том, что символ не найден
    push ds
;подставляем cs вместо ds для функции 09h (int21h)
    push cs
    pop ds
    out_str err_mes
    pop ds
    jmp num_exit               ;выход из макроса
find:                          ;совпали
                               ;считаем количество символов в строке:
    pop ax                     ;восстановим адрес начала строки
    sub di,ax                  ;(di)=(di)-(ax)
    xchg di,ax                 ; (di) < > (ax)
    sub al,3                   ;корректировка на служебные
                               ;символы - 10, 13, "$"
    aam                        ;в al две упакованные BCD-цифры
                               ;результата подсчета
    or  ax 3030h               ;преобразование результата в код ASCII
    mov cs:num,ah
    mov cs num+1,al
;вывести elem на экран
    push ds
;подставляем cs вместо ds для функции 09h (int21h)
    push cs
    pop ds
    out_str elem
    pop ds
num_exit:
    push di
    push ax
    push cx
    push es
    endm
data    segment para public "data"
msg_1   db  "Строка_1 для испытания",10,13,'$'
msg_2   db  "Строка_2 для второго испытания",10,13,'$'
data    ends
stk segment stack
```

```
    db  256 dup("?")
stk ends
code     segment para public  "code"
    assume  cs:code,ds:data,ss:stk
main    proc
    init_ds
    out_str msg_1
    num_charmsg_1
    out_str     msg_2
    num_charmsg_2
    exit
main    endp
code    ends
end main
```

В теле макроопределения можно размещать *комментарии,* и делать это особым образом. Если применить для обозначения комментария не один, как обычно, а два последовательных символа точки с запятой (;;), то при генерации макрорасширения этот комментарий будет исключен. Если по какой-то причине необходимо присутствие комментария в макрорасширении, то его нужно задавать обычным образом, то есть с помощью одного символа точки с запятой, например:

```
mes macro    messsage
... ;этот комментарий будет включен в текст листинга
... ;;этот комментарий не будет включен в текст листинга
    endm
```

Макродирективы

С помощью макросредств ассемблера можно не только частично изменять входящие в макроопределение строки, но и модифицировать сам набор этих строк и даже порядок их следования. Сделать это можно с помощью *набора макродиректив* (далее — просто директив). Их можно разделить на две группы.

- Директивы *повторения* WHILE, REPT, IRP и IRPC предназначены для создания макросов, содержащих несколько идущих подряд одинаковых последовательностей строк. При этом возможна частичная модификация этих строк.
- Директивы *управления процессом генерации макрорасширений* EXITM и GOTO предназначены для управления процессом формирования макрорасширения из набора строк соответствующего макроопределения. С помощью этих директив можно как исключать отдельные строки из макрорасширения, так и вовсе прекращать процесс генерации. Директивы EXITM и GOTO обычно используются вместе с условными директивами компиляции, поэтому они будут рассмотрены вместе с ними.

Директивы WHILE и REPT

Директивы WHILE и REPT применяются для повторения определенное количество раз некоторой последовательности строк. Эти директивы имеют следующий синтаксис:

```
WHILE константное_выражение
последовательность_строк
ENDM
REPT константное_выражение
последовательность строк
ENDM
```

Обратите внимание на то, что последовательность повторяемых строк в обеих директивах ограничена директивой **ENDM**.

При использовании директивы WHILE макрогенератор транслятора будет повторять последовательность строк до тех пор, пока значение **константное_выражение** не станет равным нулю. Это значение вычисляется каждый раз перед очередной итерацией цикла повторения (то есть значение **константное_выражение** в процессе макрогенерации должно подвергаться изменению внутри последовательности строк).

Директива REPT, подобно директиве WHILE, повторяет последовательность строк столько раз, сколько это определено значением **константное_выражение**. Отличие этой директивы от WHILE состоит в том, что она автоматически уменьшает на единицу значение **константное_выражение** после каждой итерации. В качестве примера рассмотрим листинг 14.3. В нем демонстрируется применение директив WHILE и REPT для резервирования области памяти в сегменте данных. Имя идентификатора и длина области задаются в качестве параметров соответствующих макросов def_sto_1 и def_sto_2. Заметьте, что счетчик повторений в директиве REPT уменьшается автоматически после каждой итерации цикла. Проанализируйте результат трансляции листинга 14.3.

Листинг 14.3. Использование директив повторения

```
;prg_14_3.asm
def_sto_1   macro   id_table,ln:=<5>
;макрос резервирования памяти длиной len
;Используется WHILE
id_table    label   byte
len=ln
while   len
    db  0
len=len-1
endm
endm
def_sto_2   macro   id_table,len
;макрос резервирования памяти длиной len
;Используется REPT
id_table    label   byte
rept    len
    db  0
endm
endm
data    segment para public 'data'
def_sto_1   tab_1,10
def_sto_2   tab_2,10
data    ends
init_ds macro
;Макрос настройки ds на сегмент данных
    mov ax,data
    mov ds,ax
endm
exit    macro
;макрос конца программы
    mov ax,4c00h
    int     21h
endm
code    segment para public "code"
assume  cs:code,ds:data
main    proc
```

```
init_ds
exit
main    endp
code    ends
end main
```

Таким образом, директивы REPT и WHILE удобно применять для «размножения» в тексте программы последовательности одинаковых строк без внесения в эти строки каких-либо изменений на этапе трансляции. В частности, эти директивы можно использовать при построении элементов списочных структур [8]. Следующие две директивы, IRP и IRPC, делают процесс «размножения» более гибким, позволяя модифицировать на каждой итерации некоторые элементы последовательности строк.

Директива IRP

Директива IRP имеет следующий синтаксис:

```
IRP формальный_аргумент,<строка_символов_1. . . .строка_символов_n>
последовательность_строк
ENDM
```

Действие данной директивы заключается в том, что она повторяет последовательность строк *n* раз, то есть столько раз, сколько строк символов заключено в угловые скобки во втором операнде директивы IRP. Но это еще не все. Повторение последовательности строк сопровождается заменой в этих строках формального аргумента очерёдной строкой символов из второго операнда. Так, при первой генерации последовательности строк формальный аргумент в них заменяется первой строкой символов (то есть аргументом `строка_символов_1`). Если есть вторая строка символов (`строка_символов_2`), это приводит к генерации второй копии последовательности строк, в которой формальный аргумент заменяется второй строкой символов. Эти действия продолжаются до последней строки символов (`строка_символов_n`) включительно.

К примеру, рассмотрим результат определения в программе такой конструкции:

```
irp  ini,<1,2,3,4,5>
db  ini
endm
```

Макрогенератором будет сгенерировано следующее макрорасширение:

```
db  1
db  2
db  3
db  4
db  5
```

Директива IRPC

Директива IRPC имеет следующий синтаксис:

```
IRPC формальный_аргумент,строка_символов
последовательность строк
ENDM
```

Действие данной директивы подобно действию директивы IRP, но отличается тем, что она на каждой очередной итерации заменяет формальный аргумент очередным символом из строки символов. Понятно, что количество повторений

последовательности строк будет определяться количеством символов в строке символов. К примеру,

```
irpc    rg,abcd
push    rg&x
endm
```

В процессе макрогенерации эта директива развернется в следующую последовательность строк:

```
push    ax
push    bx
push    cx
push    dx
```

Если строка символов, задаваемая в директиве IRP, содержит спецсимволы вроде точек и запятых, то она должна быть заключена в угловые скобки: <ab,,cd>.

Директивы условной компиляции

Последний тип макросредств — директивы *условной компиляции.* Существует два вида этих директив:

- *директивы компиляции по условию* позволяют проанализировать определенные условия в ходе генерации макрорасширения и при необходимости изменить этот процесс;
- *директивы генерации ошибок по условию* контролируют ход генерации макрорасширения с целью генерации или обнаружения определенных ситуаций, которые могут интерпретироваться как ошибочные.

С этими директивами применяются упомянутые ранее директивы управления процессом генерации *макрорасширений* EXITM и GOTO.

Директива EXITM не имеет операндов, она немедленно прекращает процесс генерации макрорасширения, как только встречается в макроопределении. Это дает возможность сократить объем исходного кода путем удаления неиспользуемых команд.

Директива GOTO **имя_метки** переводит процесс генерации макроопределения в другое место, прекращая тем самым последовательное разворачивание строк макроопределения. Метка, на которую передается управление, имеет специальный формат:

```
:имя_метки
```

Примеры применения этих директив будут приведены далее.

Директивы компиляции по условию

Директивы компиляции по условию предназначены для выборочной трансляции фрагментов программного кода. Это означает, что в макрорасширение включаются не все строки макроопределения, а только те, которые удовлетворяют определенным условиям. Какие конкретно условия должны быть проверены, определяется типом условной директивы. Введение в язык ассемблера этих директив значительно расширяет его возможности. Всего имеются 10 типов условных директив компиляции. Их логично попарно объединить в четыре группы:

* IF и IFE — условная трансляция по результату вычисления логического выражения;
* IFDEF и IFNDEF — условная трансляция по факту определения символического имени;
* IFB и IFNB — условная трансляция по факту определения фактического аргумента при вызове макрокоманды;
* IFIDN, IFIDNI, IFDIF и IFDIFI — условная трансляция по результату сравнения строк символов.

Условные директивы компиляции имеют общий синтаксис и применяются в составе следующей синтаксической конструкции:

```
IFxxx   логическое_выражение_или_аргументы
фрагмент_программы_1
    ELSE
    фрагмент_программы_2
ENDIF
```

Заключение некоторых фрагментов текста программы (фрагмент_программы_1 и фрагмент_программы_2) между директивами IFxxx, ELSE и ENDIF приводит к их выборочному включению в объектный модуль. Какой именно из этих фрагментов будет включен в объектный модуль, зависит от конкретного типа условной директивы, задаваемого значением xxx, и значения условия, определяемого операндом (операндами) условной директивы логическое_выражение_или_аргумент(ы).

Директивы IF и IFE

Синтаксис директив IF и IFE следующий:

```
IF(E)   логическое_выражение
фрагмент_программы_1
    ELSE
фрагмент_программы_2
ENDIF
```

Обработка этих директив макроассемблером заключается в вычислении логического выражения и включении в объектный модуль первого (фрагмент_программы_1) или второго (фрагмент_программы_2) фрагмента программы в зависимости от того, в какой директиве (IF или IFE) это выражение встретилось.

* Если в директиве IF логическое выражение истинно, то в объектный модуль помещается первый фрагмент программы. Если логическое выражение ложно, то при наличии директивы ELSE в объектный код помещается второй фрагмент программы. Если же директивы ELSE нет, то вся часть программы между директивами IF и ENDIF игнорируется, и в объектный модуль ничего не включается. Кстати сказать, понятие истинности и ложности значения логического выражения весьма условно. Ложным оно будет считаться, если его значение равно нулю, а истинным — при любом значении, отличном от нуля.
* Директива IFE аналогично директиве IF анализирует значение логического выражения. Но теперь для включения первого фрагмента программы в объектный модуль требуется, чтобы логическое выражение было ложным.

Директивы IF и IFE очень удобно использовать для изменения текста программы в зависимости от некоторых условий. К примеру, составим макрос для опре-

деления в программе области памяти длиной не более 50 и не менее 10 байт (листинг 14.4).

Листинг 14.4. Использование условных директив IF и IFE

```
<1>   ;prg_14_4.asm
<2>   masm
<3>   model   small
<4>   stack   256
<5>   def_tab_50  macro   len
<6>   if  len GE 50
<7>   GOTO exit
<8>   endif
<9>   if  len LT 10
<10>  :exit
<11>  EXITM
<12>  endif
<13>  rept  len
<14>      db  0
<15>  endm
<16>  endm
<17>  .data
<18>  def_tab_50  15
<19>  def_tab_50  5
<20>  .code
<21>  main:
<22>      mov ax,@data
<23>      mov ds,ax
<24>      exit:
<25>      mov ax,4c00h
<26>      int 21h
<27>  end main
```

Введите и оттранслируйте листинг 14.4. Не забывайте о том, что условные директивы действуют только на шаге трансляции, и поэтому результат их работы можно увидеть лишь после макрогенерации, то есть в листинге программы. В нем вы обнаружите, что в результате трансляции строка 18 листинга 14.4 развернется в пятнадцать нулевых байтов, а строка 19 оставит макрогенератор совершенно равнодушным, так как значение фактического операнда в строках 6 и 9 будет ложным. Обратите внимание на то, что для обработки реакции на ложный результат анализа в условной директиве мы использовали макродирективы EXITM и GOTO. Наверное, в данном случае можно было бы составить более оптимальный вариант макрокоманды для резервирования некоторого пространства памяти в сегменте данных, а данный способ выбран, исходя из учебных целей.

Другой интересный и полезный вариант применения директив IF и IFE — отладочная печать. Суть здесь в том, что в процессе отладки программы почти всегда возникает необходимость динамически отслеживать состояние определенных программно-аппаратных объектов, в качестве которых могут выступать переменные, регистры процессора и т. п. После этапа отладки отпадает необходимость в таких диагностических сообщениях. Для их устранения приходится корректировать исходный текст программы, после чего подвергать ее повторной трансляции. Но есть более изящный выход. Можно определить в программе некоторую переменную, к примеру debug, и использовать ее совместно с условными директивами IF или IFE:

```
...
debug   equ  1
...
```

```
<4>     .code
<5>     ...
<6>     if  debug
<7>     ;любые команды и директивы ассемблера
<8>     ;(вывод на печать или монитор)
<9>     endif
```

На время отладки и тестирования программы вы можете заключить отдельные участки кода в своеобразные операторные скобки в виде директив IF и ENDIF (строки 6-9 последнего фрагмента), реагирующие на значение логической переменной debug. При значении debug = 0 транслятор полностью проигнорирует текст внутри этих условных операторных скобок; при debug = 1, наоборот, будут выполнены все действия, описанные внутри них.

Директивы IFDEF и IFNDEF

Синтаксис директив IFDEF и IFNDEF следующий:

```
IF(N)DEF     символическое_имя
фрагмент_программы_1
   ELSE
фрагмент_программы_2
ENDIF
```

Данные директивы позволяют управлять трансляцией фрагментов программы в зависимости от того, определено или нет в программе некоторое символическое имя.

- Директива IFDEF проверяет, описано или нет в программе символическое имя, и если это так, то в объектный модуль помещается первый фрагмент программы (фрагмент_программы_1). В противном случае при наличии директивы ELSE в объектный код помещается второй фрагмент программы (фрагмент_программы_2). Если же директивы ELSE нет (и символическое имя в программе не описано), то вся часть программы между директивами IF и ENDIF игнорируется и в объектный модуль не включается.
- Действие IFNDEF обратно действию IFDEF. Если символического имени в программе нет, то транслируется первый фрагмент программы. Если оно присутствует, то при наличии ELSE транслируется второй фрагмент программы. Если ELSE отсутствует, а символическое имя в программе определено, то часть программы, заключенная между IFNDEF и ENDIF, игнорируется.

В качестве примера рассмотрим ситуацию, когда в объектный модуль программы должен быть включен один из трех фрагментов кода в зависимости от значения некоторого идентификатора switch:

- если switch = 0, то сгенерировать фрагмент для вычисления выражения $y = x \cdot 2^n$;
- если switch = 1, то сгенерировать фрагмент для вычисления выражения $y = x/2^n$;
- если идентификатор switch не определен, то ничего не генерировать.

Соответствующий фрагмент исходной программы может выглядеть так:

```
ifndef  sw       ;если sw не определено, то выйти из макроса
EXITM
else             ;иначе - на вычисление
    mov cl,n
    ife sw
        sal x,cl ;умножение на степень 2
```

```
                ;сдвигом влево
   else
      sar x,cl;деление на степень 2
                ;сдвигом вправо
   endif
endif
```

Как видим, эти директивы логически связаны с директивами IF и IFE, то есть их можно применять в тех же самых случаях, что и последние. Есть еще одна интересная возможность использования этих директив. В главе 6 мы обсуждали формат командной строки TASM и говорили о ключах, которые можно в ней задавать. Далее приведен один из них (см. приложение В, http://www.piter.com/download):

```
/dимя_идентификатора[=значение]
```

Использование этого ключа дает возможность управлять значением идентификатора прямо из командной строки, не меняя при этом текста программы. В качестве примера рассмотрим листинг 14.5. В этом коде мы пытаемся с помощью макроса контролировать процесс резервирования и инициализации некоторой области памяти в сегменте данных.

Листинг 14.5. Инициализация значения идентификатора из командной строки

```
   ;prg_14_5.asm
   masm
   model    small
   stack   256
   def_tab_50   macro   len
   ifndef   len
      display "size_m не определено, задайте значение 10<size_m<50"
      exitm
   else
   if  len GE 50
      GOTO exit
   endif
   if  len LT 10
   :exit
   EXITM
   endif
   rept  len
      db  0
   endm
endif
endm
;size_m=15
.data
def_tab_50   size_m

.code
main:
   mov ax,@data
   mov ds,ax
   exit:
   mov ax,4c00h
   int 21h
end main
```

Запустив этот пример на трансляцию, вы получите сообщение о том, что забыли определить значение переменной size_m. Исправить эту ошибку можно одним из двух способов:

- в начале исходного текста программы определите значение этой переменной с помощью псевдооператоа EQU:

  ```
  size_m equ 15
  ```

- запустите программу на трансляцию командной строкой вида

  ```
  tasm /dsize_m=15 /zi prg_13_2,,,
  ```

В листинге 14.5 мы использовали еще одну возможность транслятора — директиву DISPLAY, с помощью которой можно формировать пользовательское сообщение в процессе трансляции программы. Директива DISPLAY будет рассмотрена в конце данной главы.

Директивы IFB и IFNB

Синтаксис директив IFB и IFNB следующий:

```
IF(N)B  аргумент
фрагмент_программы_1
   ELSE
фрагмент_программы_2
ENDIF
```

Данные директивы используются для проверки фактических параметров, передаваемых в макрос. При вызове макрокоманды они анализируют значение аргумента и в зависимости от того, равно оно пробелу или нет, транслируется либо первый фрагмент программы (фрагмент_программы_1), либо второй (фрагмент_программы_2). Какой именно фрагмент будет выбран, зависит от кода директивы.

- Директива IFB проверяет равенство аргумента пробелу. В качестве аргумента могут выступать имя или число. Если его значение равно пробелу (то есть фактический аргумент при вызове макрокоманды не был задан), то транслируется и помещается в объектный модуль первый фрагмент программы. В противном случае при наличии директивы ELSE в объектный код помещается второй фрагмент программы. Если же директивы ELSE нет, то при равенстве аргумента пробелу вся часть программы между директивами IFB и ENDIF игнорируется и в объектный модуль не включается.
- Действие IFNB обратно действию IFB. Если значение аргумента в программе не равно пробелу, то транслируется первый фрагмент программы. В противном случае при наличии директивы ELSE в объектный код помещается второй фрагмент программы. Если же директивы ELSE нет, то вся часть программы (при неравенстве аргумента пробелу) между директивами IFNB и ENDIF игнорируется и в объектный модуль не включается.

Типичным примером применения этих директив являются строки в макроопределении, проверяющие, указывается ли фактический аргумент при вызове соответствующей макрокоманды:

```
show     macro    reg
ifb  <reg>
display "не задан регистр"
exitm
endif
...
endm
```

Если теперь в сегменте кода вызвать макрос SHOW без аргументов, то будет выведено сообщение о том, что не задан регистр, и генерация макрорасширения прекратится директивой EXITM.

Директивы IFIDN, IFIDNI, IFDIF и IFDIFI

Директивы IFIDN, IFIDNI, IFDIF и IFDIFI позволяют не просто проверить наличие или значения аргументов макрокоманды, но и выполнить идентификацию аргументов как строк символов. Синтаксис этих директив:

```
IFIDN(I)     аргумент_1,аргумент_2
фрагмент_программы_1
    ELSE
фрагмент_программы_2
ENDIF
IFDIF(I)     аргумент_1,аргумент_2
фрагмент_программы_1
    ELSE
фрагмент_программы_2
ENDIF
```

В этих директивах проверяются аргумент_1 и аргумент_2 как строки символов. Какой именно код (фрагмент_программы_1 или фрагмент_программы_2) будет транслироваться по результатам сравнения, зависит от кода директивы. Наличие двух пар этих директив объясняется тем, что они позволяют учитывать либо не учитывать различие строчных и прописных букв. Так, директивы IFIDNI и IFDIFI игнорируют это различие, а IFIDN и IFDIF — учитывают.

- Директива IFIDN(I) сравнивает символьные значения аргумент_1 и аргумент_2. Если результат сравнения положительный, то транслируется и помещается в объектный модуль первый фрагмент программы. В противном случае при наличии директивы ELSE, в объектный код помещается второй фрагмент программы. Если же директивы ELSE нет, то вся часть программы между директивами IFIDN(I) и ENDIF игнорируется и в объектный модуль не включается.
- Действие IFDIF(I) обратно действию IFIDN(I). Если результат сравнения отрицательный (строки не совпадают), транслируется первый фрагмент программы. В противном случае все происходит аналогично рассмотренным ранее директивам.

Как мы уже упоминали ранее, эти директивы удобно применять для проверки фактических аргументов макрокоманд. К примеру, проверим, какой из регистров — al или ah — передан в макрос в качестве параметра (проверка проводится без учета различия строчных и прописных букв):

```
show    macro   rg
ifdifi  <al>,<rg>
goto    M_al
else
ifdifi  <ah>,<rg>
goto    M_ah
else
exitm
endif
endif
:M_al
...
```

```
:M_ah
...
endm
```

Вложенность директив условной трансляции

Как мы неоднократно видели в приведенных ранее примерах, транслятор TASM допускает вложенность условных директив компиляции. Более того, так как вложенность требуется довольно часто, TASM предоставляет набор дополнительных директив формата ELSEIFxxx, которые заменяют последовательность идущих подряд директив ELSE и IFxxx в структуре

```
IFxxx
;...
    ELSE
    IFxxx
;
ENDIF
ENDIF
```

Эту последовательность условных директив можно заменить эквивалентной последовательностью дополнительных директив

```
IFxxx
;...
    ELSEIFxxx
;...
ENDIF
```

Символы xxx в ELSExxx говорят о том, что каждая из директив — IF, IFB, IFIDN и т. д. — имеет соответствующую директиву ELSEIF, ELSEIFB, ELSEIFIDN и т. д. В конечном итоге это улучшает читабельность кода. В последнем примере фрагмента макроса, проверяющем, имя какого регистра было передано в макрос, наблюдается подобная ситуация. Последовательности ELSE и IFDIFI можно записать так, как в строке 4:

```
<1> showmacro rg
<2> ifdifi <al>,<rg>
<3> goto M_al
<4> else ifdifi <ah>,<rg>
<5> goto M_ah
<6> else
<7> exitm
<8> endif
<9> :M_al
<10>    ...
<11>    :M_ah
<12>    ...
<13>    endm
```

Директивы генерации ошибок

В языке TASM есть ряд директив, называемых *директивами генерации пользовательской ошибки*. Их можно рассматривать и как самостоятельное средство, и как метод, расширяющий возможности директив условной компиляции. Они предназначены для обнаружения различных ошибок в программе, таких как неопределенные метки или пропуск параметров макроса. Директивы генерации пользовательской ошибки по принципу работы можно разделить на два типа:

- безусловные директивы генерируют ошибку трансляции без проверки каких-либо условий;

it условные директивы генерируют ошибку трансляции после проверки определенных условий.

Большинство директив генерации ошибок имеет два обозначения, хотя принцип их работы одинаков. Второе название отражает их сходство с директивами условной компиляции. При дальнейшем обсуждении такие парные директивы будут приводиться в скобках. •

Безусловная генерация пользовательской ошибки

К директивам безусловной генерации пользовательской ошибки относится только одна директива ERR (.ERR).

Данная директива, будучи вставлена в текст программы, безусловно приводит к генерации ошибки на этапе трансляции и удалению объектного модуля. Она очень полезна при использовании с директивами условной компиляции и при включении в тело макрокоманды с целью отладки. К примеру, эту директиву можно было бы вставить в ту ветвь программы (в последнем рассмотренном нами макроопределении), которая выполняется, если указанный в качестве аргумента регистр отличен от al и ah:

```
show    macro rg
ifdifi  <al>,<rg>
goto    M_al
else
ifdifi  <ah>,<rg>
goto    M_ah
else
.Err
endif
endif
...
endm
```

Если после определенного таким образом макроопределения в сегменте кода вызвать макрокоманду show с фактическим параметром, отличным от имен регистров ah или al, будет сгенерирована ошибка компиляции (с текстом «User error»), сам процесс компиляции прекращен и, естественно, объектный модуль создан не будет.

Остальные директивы являются *условными,* так как их поведение определяют некоторые условия.

Условная генерация пользовательской ошибки

Набор условий, на которые реагируют директивы условной генерации пользовательской ошибки, такой же, как и у директив условной компиляции. Поэтому и количество этих директив такое же. Принцип их работы ясен, поэтому рассматривать их мы будем очень кратко. Заметим только, что использовать большинство директив условной генерации пользовательской ошибки, как и директив условной компиляции, можно и в макроопределениях, и в любых местах программы.

Директивы .ERRB (ERRIFB) и .ERRNB (ERRIFNB)

Синтаксис директив .ERRB (ERRIFB) и .ERRNB (ERRIFNB):

```
.ERRB (ERRIFB) <имя_формального_аргумента>
.ERRNB (ERRIFNB) <имя_формального_аргумента>
```

- Директива .ERRB (ERRIFB) вызывает генерацию пользовательской ошибки, если формальный аргумент с именем **<имя_формального_аргумента>** пропущен.
- Директива .ERRNB (ERRIFNB) вызывает генерацию пользовательской ошибки, если формальный аргумент с именем **<имя_формального_аргумента>** присутствует.

Данные директивы применяются для генерации ошибки трансляции в зависимости от того, задан или нет при вызове макрокоманды фактический аргумент, соответствующий формальному аргументу в заголовке макроопределения с именем **<имя_формального_аргумента>**. По принципу действия эти директивы полностью аналогичны соответствующим директивам условной компиляции IFB и IFNB. Их обычно используют для проверки задания параметров при вызове макроса. Строка, являющаяся именем формального аргумента, должна быть заключена в угловые скобки.

К примеру, определим обязательность задания фактического аргумента, соответствующего формальному аргументу rg, в макросе show:

```
showmacro rg
;если rg в макрокоманде не будет задан,
;то завершить компиляцию
.errb   <rg>
 ;текст макроопределения
;...
endm
```

Директивы .ERRDEF (ERRIFDEF) и .ERRNDEF (ERRIFNDEF)

Синтаксис директив .ERRDEF (ERRIFDEF) и .ERRNDEF (ERRIFNDEF):

```
.ERRDEF (ERRIFDEF) символическое_имя
.ERRNDEF (ERRIFNDEF) символическое_имя
```

- Директива .ERRDEF (ERRIFDEF) генерирует пользовательскую ошибку, если указанное символическое имя определено до выдачи этой директивы в программе.
- Директива .ERRNDEF (ERRIFNDEF) генерирует пользовательскую ошибку, если указанное символическое имя не определено до момента обработки транслятором этой директивы.

Данные директивы генерируют ошибку трансляции в зависимости от того, определено или нет некоторое символическое имя в программе. Не забывайте о том, что компилятор TASM по умолчанию формирует объектный модуль за один проход исходного текста программы. Следовательно, директивы .ERRDEF (ERRIFDEF) и .ERRNDEF (ERRIFNDEF) отслеживают факт определения символического имени только в той части исходного текста, которая находится до этих директив.

Директивы .ERRDIF (ERRIFDIF) и .ERRIDN (ERRIFIDN)

Синтаксис директив .ERRDIF (ERRIFDIF) и .ERRIDN (ERRIFIDN):

```
.ERRDIF (ERRIFDIF) <строка_1>,<строка_2>
.ERRIDN (ERRIFIDN) <строка_1>,<строка_2>
```

- Директива .ERRDIF (ERRIFDIF) генерирует пользовательскую ошибку, если две строки посимвольно не совпадают. Строки могут быть символическими именами, числами или выражениями и должны быть заключены в угловые скобки. Аналогично директиве условной компиляции IFDIF, при сравнении учитывается различие прописных и строчных букв.

ж Директива .ERRIDN (ERRIFIDN) генерирует пользовательскую ошибку, если строки посимвольно идентичны. Строчные и прописные буквы воспринимаются как разные.

Для того чтобы игнорировать различия строчных и прописных букв, существуют аналогичные директивы:

```
ERRIFDIFI <строка_1>,<строка_2>
ERRIFIDNI <строка_1>,<строка_2>
```

я Директива ERRIFDIFI аналогична директиве ERRIFDIF, но игнорируется различие строчных и прописных букв при сравнении строк <строка_1> и <строка_2>.

9 Директива ERRIFIDNI аналогична директиве ERRIFIDN, но игнорируется различие строчных и прописных букв при сравнении строк <строка_1> и <строка_2>.

Данные директивы, как и соответствующие им директивы условной компиляции, удобно применять для проверки передаваемых в макрос фактических параметров.

Директивы .ERRE (ERRIFE) и .ERRNZ (ERRIF)

Синтаксис директив .ERRE (ERRIFE) и .ERRNZ (ERRIF):

```
.ERRE (ERRIFE) константное_выражение
.ERRNZ (ERRIF) константное_выражение
```

в Директива .ERRE (ERRIFE) вызывает пользовательскую ошибку, если константное выражение ложно (равно нулю). Вычисление константного выражения должно приводить к абсолютному значению, и это выражение не может содержать ссылок вперед.

▒ Директива .ERRNZ(ERRIF) вызывает пользовательскую ошибку, если константное выражение истинно (не равно нулю). Вычисление константного выражения должно приводить к абсолютному значению и не может содержать ссылок вперед.

Константные выражения в условных директивах

Как вы успели заметить, во многих условных директивах в формировании условия участвуют выражения. Результат вычисления этого выражения обязательно должен быть константой. Хотя его компонентами могут быть и символические параметры, но их сочетание в выражении должно давать абсолютный результат. К примеру,

```
...
.data
mas db  ...
len dd  ...
...
.code
...
.erre   (len-mas) lt 10 ;генерация ошибки, если длина
                        ;области mas меньше 10 байт
...
```

Кроме того, константное выражение не должно содержать компоненты, которые транслятор еще не обработал к тому месту программы, где находится условная директива. Также мы отметили, что логические результаты «истина» и «ложь» являются условными в том смысле, что ноль соответствует логическому результа-

ту «ложь», а любое ненулевое значение — «истине». Однако в языке ассемблера существуют операторы, которые позволяют сформировать и «чисто логический» результат. Это так называемые операторы отношений, выражающие отношение двух значений или константных выражений. В контексте условных директив вместе с операторами отношений можно рассматривать и логические операторы. Результатом работы тех и других может быть одно из двух значений:

- истина — число, которое содержит двоичные единицы во всех разрядах;
- ложь — число, которое содержит двоичные нули во всех разрядах.

Операторы, которые можно применять в выражениях условных директив и которые формируют логические результаты, приведены в табл. 14.1 и 14.2.

Таблица 14.1. Операторы отношений

Оператор отношения	Синтаксис	Результат
EQ (equal) — равно	выражение_1 EQ выражение_2	Истина, если выражение_1 равно выражение_2
NE (not equal) — не равно	выражение_1 NE выражение_2	Истина, если выражение_1 не равно выражение_2
LT (less than) — меньше	выражение_1 LT выражение_2	Истина, если выражение_1 меньше выражение_2
LE (less or equal) — меньше или равно	выражение_1 LE выражение 2	Истина, если выражение_1 меньше или равно выражение 2
GT (greater than) - больше	выражение_1 GT выражение_2	Истина, если выражение_1 больше выражение_2
GE (greater or equal) — больше или равно	выражение_1 GE выражение_2	Истина, если выражение_1 больше или равно выражение_2

Таблица 14.2. Логические операторы

Логический оператор	Синтаксис	Результат
NOT — логическое отрицание	NOT выражение	Истина, если выражение ложно; ложь, если выражение истинно
AND — логическое И	выражение_1 AND выражение_2	Истина, если выражение_1 и выражение_2 истинны
OR — логическое ИЛИ	выражение_1 OR выражение 2	Истина, если выражение_1 или выражение 2 истинны
XOR — исключающее ИЛИ	выражение_1 XOR выражение_2	Истина, если выражение_1 равно логическому отрицанию выражения_2

Дополнительные средства управления трансляцией

TASM предоставляет средства для вывода текстового сообщения во время трансляции программы — директивы DISPLAY и %OUT. С их помощью можно при необходимости следить за ходом трансляции. К примеру,

```
display недопустимые аргументы макрокоманды
...
%out    недопустимое имя регистра
```

В результате обработки этих директив на экран (стандартный выход) будут выведены тексты сообщений. Если эти директивы использовать совместно с директивами условной компиляции, то, к примеру, можно отслеживать путь, по которому осуществляется трансляция исходного текста программы.

Можно предложить читателю уже с этого момента начать формировать набор полезных в его практической работе макрокоманд. В качестве основы вы можете взять файл mac.inc, который находится в каталоге данной главы среди файлов, прилагаемых к книге[1]. В дальнейшем, если в этом возникнет необходимость, вы будете самостоятельно дополнять его вашими макросами. Использовать макроопределения из этого файла очень просто: достаточно включить в нужном месте вашей программы строку с директивой include, в результате в ваш файл будут вставлены строки из файла, указанного в качестве операнда этой директивы.

Основная задача этой книги — научить вас программировать на языке ассемблера. Как вы уже успели понять, нельзя изучать этот язык в отрыве от рассмотрения процессов, происходящих во время выполнения программы на компьютере. Одно из средств изучения таких процессов — отладчик. Однако он решает проблему глобально, что нужно далеко не всегда. Тем более, как мы увидим далее, что возможности отладчика не безграничны. Поэтому необходимо иметь более универсальное средство, которое позволило бы «подглядывать» за содержимым регистра или области памяти динамически, во время выполнения программы. Для этого разработаем еще один макрос и назовем его, к примеру, show. Его аргументом может быть один из четырех регистров — AL, AH, AX, EAX. С помощью этого макроса можно визуализировать содержимое любого из доступных регистров или области памяти длиной до 32 битов. Для этого достаточно лишь переслать содержимое нужного объекта (регистра или ячейки памяти) с учетом его размера в один из регистров AL, AH, AX, EAX. Имя одного из этих регистров указывается затем в качестве фактического аргумента макрокоманды show. Второй аргумент этого макроса — позиция на экране. Задавая определенные значения, мы можем судить о том, какая именно макрокоманда show сработала. Еще одна немаловажная особенность данного макроса — в его возможности работать как в реальном, так и в защищенном режимах. Распознавание текущего режима работы процессора выполняется автоматически. Текст макроопределения show довольно велик и по этой причине находится среди прилагаемых к книге файлов в каталоге данной главы. Пример использования этого макроса приведен в листинге 14.6.

Листинг 14.6. Пример использования макроса show

```
;prg_14_6.asm
MASM
MODEL   small
STACK   256
.486p
Include show.inc
.data
```

[1] Все прилагаемые к книге файлы можно найти по адресу http://www.piter.com/download. — *Примеч. ред.*

```
pole     dd  3cdf436fh
.code
main:
    mov ax,@data
    mov ds,ax
    xor  ax,ax
    mov ax,1f0fh
    show   al,0
    show ah,160
    show ax,320
    mov eax,pole
    show eax,480
    exit:
    mov ax,4c00h
    int 21h
end main
```

Посвятить время рассмотрению этого макроса полезно еще и потому, что при его разработке было использовано большинство средств, обсуждавшихся в этой главе.

Директивы управления файлом листинга

Ассемблер предоставляет ряд директив для управления содержимым файла листинга. Далее приведена их общая характеристика.

Общие директивы управления листингом призваны управлять видом файла листинга. Все директивы являются парными, это означает, что если одна директива что-то разрешает, то другая, наоборот, запрещает.

- Директивы %LIST и %NOLIST (.LIST и .XLIST) определяют необходимость вывода в файл листинга всех строк исходного кода (по умолчанию). Для запрета вывода в файл листинга всех строк исходного кода необходимо использовать директивы .XLIST или %NOLIST. В тексте программы их можно применять произвольное количество раз, при этом очередная директива отменяет действие предыдущей.
- Директивы %CTLS и %NOCTLS управляют выводом в файл листинга самих директив управления листингом, в то время как директивы %LIST и %NOLIST (.LIST и .XLIST) влияют на полноту представления исходного кода в целом.
- Директивы %SYMS и %NOSYMS определяют включение (%SYMS) или не включение (%NOSYMS) таблицы идентификаторов в файл листинга.

Директивы вывода текста включаемых файлов %INCL и %NOINCL регулируют включение в файл листинга текста включаемых файлов (по директиве INCLUDE). По умолчанию включаемые файлы записываются в файл листинга. Директива %NOINCL запрещает вывод в файл листинга всех последующих включаемых файлов, пока вывод снова не будет разрешен директивой %INCL.

Директивы вывода блоков условного ассемблирования регулируют включение блоков условной компиляции в листинг программы.

- Директива %CONDS (.LFCOND) заставляет ассемблер выводить в файл листинга все операторы условных блоков (в том числе с условием false). Директива %NOCONDS (.SFCONDS) запрещает вывод в файл листинга блоков условного ассемблирования с условием false.

11 Зак. 256

- Директива .TFCOND переключает режимы вывода %CONDS (.LFCOND) и %NOCONDS (.SFCONDS). Эту директиву можно использовать как отдельно, так и совместно с директивами .LFCOND и .SFCONDS. Первая директива .TFCOND, которую обнаруживает TASM, разрешает вывод в файл листинга всех блоков условного ассемблирования. Следующая директива .TFCOND запретит вывод этих блоков. С директивой .TFCOND можно использовать ключ /x командной строки транслятора TASM — в этом случае блоки условного ассемблирования будут сначала выводиться в файл листинга, но первая же директива .TFCOND запретит их вывод.

Директивы вывода макрорасширений, как явствует из названия, управляют выводом макрорасширений:

- Директива %MACS (.LALL) разрешает вывод в файл листинга всех макрорасширений.
- Директива %NOMACS (.SALL) запрещает вывод всех операторов макрорасширения в файл листинга.
- В трансляторе MASM можно использовать директиву .XALL, позволяющую выводить в листинг только те макрорасширения, которые генерируют код или данные.

Итоги

- Преимущества языка ассемблера связаны, в частности, с макросредствами. Как говорят, если бы макросредств в нем не было, то их нужно было бы придумать.
- Макросредства — это основные инструменты модификации текста программы на этапе ее трансляции. Принцип работы макросредств основан на препроцессорной обработке, которая заключается в том, что текст, поступающий на вход транслятора, перед собственно компиляцией подвергается преобразованию и может значительно отличаться от синтаксически правильного текста, воспринимаемого компилятором. Роль препроцессора в трансляторе TASM выполняет макрогенератор.
- Для того чтобы макрогенератор мог выполнить свою работу, текст программы должен удовлетворять определенным требованиям. Макрогенератору необходимо сообщить, на какие элементы исходного текста он должен реагировать и какие действия должны быть произведены. Можно выделить несколько типов таких элементов.
- Псевдооператоры EQU и = предназначены для присвоения некоторому выражению символического имени или идентификатора. Эти действия выполняет макрогенератор, заменяя в последующем тексте программы символические имена из правой части этих операторов строками из левой.
- Макрокоманда — строка в исходной программе, которой соответствует специальный блок — макроопределение. Макрокоманда может иметь аргументы, с помощью которых можно изменять текст макроопределения. Макрогенератор, встречая макрокоманду в тексте программы, корректирует текст соответствующего макроопределения, исходя из аргументов этой макрокоманды, и вставля-

ет его в текст программы вместо данной макрокоманды. Процесс такого замещения называется макрогенерацией.

- Условные директивы компиляции позволяют не просто модифицировать отдельные строки программы, но и, исходя из определенных условий, управлять включением в загрузочный модуль отдельных фрагментов программы. Эти директивы наиболее эффективны для работы с аргументами, передаваемыми при макрогенерации в макроопределения из макрокоманд, хотя отдельные директивы есть смысл применять и вне макроопределений в любом месте программы.
- Директивы генерации ошибок, подобно условным директивам, позволяют анализировать определенные условия в процессе трансляции программы и генерировать ошибку по результатам анализа.
- Для удобства формирования файла листинга компиляторы ассемблера предоставляют в распоряжение программиста ряд директив. С помощью этих директив можно довольно гибко изменять формат и полноту информации, выводимой ассемблером в файл листинга.

Глава 15
Модульное программирование

- Основы структурного программирования
- Средства ассемблера для поддержки структурного программирования
- Процедуры и организация связей между процедурами на языке ассемблера
- Директива INVOKE (MASM)
- Связь между программами на языках высокого уровня и программами на ассемблере

Мы неоднократно подчеркивали один из существенных недостатков программ на языке ассемблера, а значит, и самого языка, — недостаточную наглядность. По прошествии даже небольшого времени программисту порой бывает трудно разобраться в деталях им же написанной программы. А о чужой программе и говорить не приходится. Если в ней нет хотя бы минимальных комментариев, то разобраться с тем, что она делает, довольно трудно. Причины этого тоже понятны — при написании программы на языке ассемблера человек должен запрограммировать самые элементарные действия или операции. При этом он должен учитывать и контролировать состояние большого количества данных. Из-за элементарности программируемых операций реализация одного и того же алгоритма может быть произведена по крайней мере несколькими способами. Эта неоднозначность влечет за собой непредсказуемость, что и затрудняет процесс обратного восстановления исходного алгоритма по ассемблерному коду.

По мере накопления опыта эти проблемы частично снимаются. Но одного опыта мало. Ситуация усугубляется, если работает коллектив разработчиков. Тут уже

нужны специальные средства. Далее перечислены организационные и программные (предоставляемые ассемблером) мероприятия, позволяющие хотя бы частично снять остроту этой проблемы.

- Документирование программистом своей работы и ее результатов. Делается это в первую очередь путем комментирования строк исходного текста программы. При этом комментарии должны коротко, но точно выражать то, что делает данная программа в целом, выделять ее наиболее важные фрагменты и особенности применения отдельных команд. В конечном итоге, комментирование облегчает понимание сути программы, но все-таки полностью не снимает проблему.
- Упрощение кода программы путем замены сложных фрагментов более понятным кодом. Для этого, в частности, можно использовать рассмотренные нами макрокоманды.
- Использование при разработке программных проектов достижений современных технологий программирования.

К настоящему моменту наиболее популярными и жизнеспособными оказались две технологии программирования: структурная и объектно-ориентированная.

Последние версии пакетов TASM и MASM языка ассемблера поддерживают объектно-ориентированное программирование, но реализация его довольно сложна и требует отдельного рассмотрения. Типичному процессу написания программы на ассемблере больше всего удовлетворяют концепции структурного программирования. Можно даже сказать, что для процессорной архитектуры IA-32 эти концепции поддерживаются на аппаратном уровне с помощью таких архитектурных механизмов, как сегментация памяти и аппаратная реализация команд передачи управления. На программном уровне поддержка заключается, в основном, в соответствующих средствах конкретного компилятора, в частности, такие средства имеют компиляторы TASM и MASM. Данная глава посвящен этим программно-аппаратным средствам.

Структурное программирование

Структурное программирование — методология программирования, базирующаяся на системном подходе к анализу, проектированию и реализации программного обеспечения. Эта методология зародилась в начале 70-х гг. и оказалась настолько жизнеспособной, что и до сих пор является основной в большом количестве проектов. Ее основу составляет *концепция модульного программирования.*

Концепцию модульного программирования можно сформулировать в виде нескольких понятий и положений. Основа концепции модульного программирования — ***модуль***, который является продуктом процесса разбиения большой задачи на ряд более мелких функционально самостоятельных подзадач. Этот процесс называется ***функциональной декомпозицией задачи.*** Каждый модуль в функциональной декомпозиции представляет собой «черный ящик» с одним входом и одним выходом. Модули связаны между собой только входными и выходными данными.

Модульный подход позволяет безболезненно производить модернизацию программы в процессе ее эксплуатации и облегчает ее сопровождение. Дополнительно модульный подход позволяет разрабатывать части программ одного проекта на разных языках программирования, после чего с помощью компоновочных средств объединять их в единый загрузочный модуль.

Основная цель модульного подхода — простота и ясность реализуемых решений. Если вы затрудняетесь четко сформулировать назначение модуля, это говорит о том, что один из предыдущих этапов декомпозиции задачи был проведен недостаточно качественно. В этом случае необходимо вернуться на один или более шаг назад и еще раз проанализировать задачу этапа, вызвавшего проблему. Это, возможно, приведет к появлению дополнительных модулей в процессе функциональной декомпозиции.

При наличии в проекте сложных мест их нужно подробно документировать с помощью продуманной системы комментариев. Комментарии должны быть ясными в смысле отражения реализуемых идей и подчиняться некоторой системе — частной, если разработчик — частное лицо, или корпоративной, действующей внутри некоторого организованного коллектива разработчиков. По поводу системы комментирования можно еще раз напомнить несколько широко известных идей. Во-первых, желательно назначение всех переменных модуля описывать комментариями по мере их определения. Во-вторых, исходный текст модуля должен иметь заголовок, отражающий назначение модуля и его внешние связи. Этот заголовок можно назвать ***интерфейсной частью модуля.*** В интерфейсной части с использованием комментариев нужно поместить информацию о назначении модуля, об особенностях функционирования, о входных и выходных аргументах, об использовании внешних модулей и переменных, а также о разработчике — для защиты авторских прав.

В ходе разработки программы следует предусматривать специальные блоки операций, учитывающие реакцию на возможные ошибки в данных или в действиях пользователя. Это очень важный момент, означающий, что в алгоритме программы не должно быть тупиковых ветвей, в результате работы которых программа «виснет» и перестает отвечать на запросы пользователя. Любые непредусмотренные действия пользователя должны приводить к генерации ошибочной ситуации или к предупреждению о возможности возникновения такой ситуации.

Из этих положений видно, какое большое значение придается организации *управляющих* и *информационных связей* между структурными единицами программы (модулями), совместно решающими одну или несколько больших задач. Применительно к языку ассемблера можно рассматривать несколько форм организации управляющих связей.

Si Макроподстановки позволяют изменять исходный текст программы в соответствии с некоторыми предварительно описанными параметризованными объектами. Эти объекты имеют формальные аргументы, замещаемые в процессе макрогенерации их фактическими аргументами. Такая форма образования структурных элементов носит некоторый предварительный характер из-за того, что процессы замены происходят на этапе компиляции и есть смысл рассматривать их только как настройку на определенные условия функционирования программы.

- Объединение в одну программу подпрограмм, написанных на ассемблере. В языке ассемблера такие подпрограммы называют *процедурами.* В отличие от макрокоманд, взаимодействие процедур осуществляется на этапе выполнения программы.
- Объединение в единый модуль на этапе компоновки подпрограмм, написанных на разных языках программирования. Эта возможность реализуется благодаря унифицированному формату объектного модуля, однозначным соглашениям о передаче аргументов и единым схемам организации памяти на этапе выполнения.
- Динамический (на этапе выполнения) вызов исполняемых модулей и динамическое подключение библиотек (DLL-файлов) для операционной системы Windows.

В качестве основных информационных связей можно выделить следующие:

- общие области памяти и общие программно-аппаратные ресурсы процессора для связи модулей;
- унифицированная передача аргументов при *вызове* модуля (эту унификацию можно представлять двояко: на уровне пользователя и на уровне конкретного компилятора);
- унифицированная передача аргументов при *возвращении управления* из модуля.

Чуть позже мы подробно рассмотрим процессы, происходящие при передаче аргументов. Сейчас в качестве некоторого итога приведенных ранее общих рассуждений перечислим средства языка ассемблера по осуществлению *функциональной декомпозиции* программы:

- макросредства;
- процедуры;
- средства компилятора ассемблера в форме директив организации оперативной памяти и ее сегментации.

Макросредства подробно рассмотрены в главе 14. Предварительное обсуждение процедур проведено при обсуждении команд передачи управления в главе 10. Но процедуры — это не просто механизм тривиальной передачи управления из одной точки программы в другую. В частности, этот механизм тесно связан со средствами компилятора, поддерживающими организацию памяти и сегментацию. Поэтому дальнейшее обсуждение будет посвящено более глубокому изучению функциональной декомпозиции программ с использованием процедур и связанных с ними средств компилятора.

Процедуры в языке ассемблера

В Ассемблере для оформления процедур как отдельных объектов существуют специальные директивы `PROC/ENDP` и машинная команда `RET` (см. главу 10). Если сравнивать процедуры и макрокоманды, то можно сказать следующее. Процедуры, так же как и макрокоманды, могут быть активизированы в любом месте программы.

Процедурам, так же как и макрокомандам, могут быть переданы некоторые аргументы. Это позволяет, имея одну копию кода в памяти, изменять ее для каждого конкретного случая использования, хотя по гибкости процедуры уступают макрокомандам.

В главе 10 нами были рассмотрены возможные варианты размещения процедур в программе:

- в начале программы (до первой исполняемой команды);
- в конце программы (после команды, возвращающей управление операционной системе);
- промежуточный вариант — тело процедуры располагается внутри другой процедуры или основной программы (в этом случае необходимо предусмотреть обход процедуры с помощью команды безусловного перехода JMP);
- в другом модуле.

Главная цель таких вариантов размещения — не допустить несанкционированной передачи управления коду процедуры. Три первых варианта относятся к случаю, когда процедуры находятся в одном сегменте кода. Мы их достаточно подробно обсудили. Что же касается последнего варианта, то он предполагает, что процедуры находятся в разных модулях. А это дает нам возможность говорить уже не об одном модуле, а о нескольких. Для реализации одной общей задачи эти модули должны быть связаны между собой по управлению и по данным. Если мы разберемся с тем, как организовать такую связь, то фактически сможем выполнить функциональную декомпозицию любой большой программы на нужное количество более мелких. В первой части главы мы рассмотрим, как организуется связь по управлению и по данным между модулями на ассемблере, а во второй части — между модулями на ассемблере и на языках высокого уровня (Pascal и C/C++).

Сначала необходимо отметить один общий для всех этих трех языков момент. Так как отдельный модуль в соответствии с концепцией модульного программирования — это функционально автономный объект, то он ничего не должен знать о внутреннем устройстве других модулей, и наоборот, другим модулям также ничего не должно быть известно о внутреннем устройстве данного модуля. Однако должны быть какие-то средства, с помощью которых можно связать модули. В качестве аналогии можно привести организацию связи (интерфейс) телевизора и видеомагнитофона через разъем типа «скарт». Связь унифицирована, то есть известно, что один контакт предназначен для видеосигнала, другой — для передачи звука и т. д. Телевизор и видеомагнитофон могут быть разными, но связь между ними одинакова. Та же идея лежит и в организации связи модулей. Внутреннее устройство модулей может совершенствоваться, они вообще могут в следующих версиях писаться на другом языке, но в процессе их объединения в единый исполняемый модуль этих особенностей не должно быть заметно. Таким образом, каждый модуль должен иметь такие средства, с помощью которых он извещал бы транслятор о том, что некоторый объект (процедура, переменная) видимым вне этого модуля. И наоборот, нужно объяснить транслятору, что некоторый объект находится вне данного модуля. Это позволит транслятору правильно сформировать машинные команды, оставив некоторые их поля не заполненными. Позднее, на этапе компо-

новки, программа TLINK (TASM) или программа компоновки языка высокого уровня произведут настройку модулей и разрешат все внешние ссылки в объединяемых модулях.

Для того чтобы объявить о подобного рода объектах, видимых извне, программа должна использовать две директивы TASM: EXTRN и PUBLIC. Директива EXTRN предназначена для объявления некоторого имени внешним по отношению к данному модулю. Это имя в другом модуле должно быть объявлено в директиве PUBLIC. Директива PUBLIC предназначена для объявления некоторого имени, определенного в этом модуле и видимого в других модулях. Синтаксис этих директив следующий:

```
extrn имя:тип ... имя:тип
public имя,...,имя
```

Здесь имя — идентификатор, определенный в другом модуле. В качестве идентификатора могут выступать:

- имена переменных, определенных директивами типа DB, DW и т. д.;
- имена процедур;
- имена констант, определенных операторами = и EQU.

Аргумент тип определяет тип идентификатора. Указание типа необходимо для того, чтобы транслятор правильно сформировал соответствующую машинную команду. Действительные адреса вычисляются на этапе редактирования, когда будут разрешаться внешние ссылки. Возможные значения типа определяются допустимыми типами объектов для этих директив:

- если имя — это имя переменной, то тип может принимать значения BYTE, WORD, DWORD, PWORD, FWORD, QWORD и TBYTE;
- если имя — это имя процедуры, то тип может принимать значения near или far;
- если имя — это имя константы, то тип должен быть abs.

Покажем принцип использования директив EXTRN и PUBLIC на схеме связи модулей 1 и 2 (листинги 15.1 и 15.2).

Листинг 15.1. Модуль 1

```
;Модуль 1
masm
.model  small
.stack  256
.data
...
.code
my_proc_1   proc
...
my_proc_1   endp
my_proc_2   proc
...
my_proc_2   endp
;объявляем процедуру my_proc_1 видимой извне
public  my_proc_1
start:
    mov ax,@data
...
end start
```

Листинг 15.2. Модуль 2

```
;Модуль 2
masm
.model  small
.stack 256
.data
...
.code
extrn   my_proc_1   ;объявляем процедуру roy_proc_l внешней
start:
    mov ax,@data
...
    call my_proc_1  ;вызов my_proc_l из модуля 1
end start
```

Рассмотренная нами схема связи — это, фактически, связь по управлению. Но не менее важно организовать информационный обмен между модулями. Рассмотрим основные способы организации такой связи.

Информационный обмен между модулями (процедурами) предполагает обмен данными. В этой связи важно понимать значение терминов *аргумент, переменная, константа.*

Аргумент — это ссылка на некоторые данные, которые требуются для выполнения возложенных на модуль функций и размещенных вне этого модуля. По аналогии с макрокомандами выделяют *формальные* и *фактические* аргументы. Исходя из этого, формальный аргумент можно рассматривать не как непосредственные данные или их адрес, а как «тару» для действительных данных, которые будут положены в нее с помощью фактического аргумента. Формальный аргумент можно рассматривать как элемент интерфейса модуля (конкретный вывод «скарта»), а фактический аргумент — это то, что фактически передается на место формального аргумента.

Переменная — это нечто, размещенное в регистре или ячейке памяти, что может в дальнейшем подвергаться изменению.

Константа — информационный объект простого типа, значение которого никогда не изменяется.

Таким образом, если некоторые данные в модуле могут подвергаться изменению, то это *переменные.* Если переменная находится за пределами модуля (процедуры) и должна быть как-то передана в него, то для модуля она является формальным *аргументом.* Значение переменной передается в модуль для замещения соответствующего параметра при помощи фактического *аргумента.* После пояснения различия понятий формального и фактического аргументов далее по тексту они будут называться обобщенно — *аргументы,* а о каком виде аргументов идет речь, станет понятно по контексту.

Если входные данные для модуля (аргументы) — переменные, то один и тот же модуль можно использовать многократно для разных наборов значений этих переменных. Но как организовать передачу значений переменных в модуль (процедуру)? При программировании на языке высокого уровня программист ограничен в выборе способов передачи аргументов теми рамками, которые для него оставляет компилятор. В языке ассемблера практически нет никаких ограничений на этот счет, и, фактически, решение проблемы передачи аргументов предоставлено программисту.

На практике используются следующие варианты передачи аргументов в модуль (процедуру):

- через регистры;
- через общую область памяти;
- через стек;
- с помощью директив EXTRN и PUBLIC.

Передача аргументов через регистры

Передача аргументов через регистры — наиболее простой в реализации способ передачи данных. Данные, переданные подобным способом, становятся доступными немедленно после передачи управления процедуре. Этот способ можно рекомендовать для передачи небольших объемов данных. Недостатки этого способа:

- небольшое число доступных для пользователя регистров;
- необходимость постоянно помнить о том, какая информация в каком регистре находится;
- ограничение размера передаваемых данных размерами регистра — если размер данных превышает 8, 16 или 32 бита, то передачу данных посредством регистров произвести нельзя и передавать приходится не сами данные, а указатели на них.

Передача аргументов через регистры широко применяется при вызове функций DOS.

В главе 14 мы обсуждали программу (см. листинг 14.2) с макрокомандой, которая подсчитывала длину строки, оканчивающуюся символом $. Для сравнения эффективности применения макрокоманд и процедур при программировании разработаем аналогичную программу (листинг 15.3), но с использованием процедуры CountSymbol, подсчитывающей количество символов в строке с конечным символом $. Процедура располагается в конце программы. Длина строки — не более 99 символов. Адрес строки передается процедуре как аргумент через регистр si. Результат подсчета возвращается в регистр BL и выводится на экран в вызывающей программе. Для вывода на экран используется прямой доступ к видеобуферу.

Листинг 15.3. Передача аргументов через регистры

```
;prg_15_3.asm
MASM
MODEL   small           ;модель памяти
STACK   256             ;размер стека
include mac.inc         ;подключение файла с макросами
.data                   ;начало сегмента данных
maskd   db  71h         ;маска вывода на экран
string  db  "Строка для подсчета $"  ;тестовая строка
mes db  "В строке string "
cnt db  2 dup ("*") ; количество символов в строке
    db " символов",10,13,'$'

.code
main    proc            ;точка входа в главную процедуру
    mov ax,@data
    mov ds,ax
```

продолжение ⇨

Листинг 15.3 *(продолжение)*

```
;загрузка адреса строки
;(для передачи смещения в процедуру)
    lea si,string
;вызов процедуры
    call CountSymbol
    mov cl,bl         ;счетчик для lods и stosw
    lea si,string     ;в si - указатель на строку
    mov ax,0b800h
    mov es,ax         ;загрузка в es адреса видеопамяти
    mov ah,maskd      ;маска вывода на экран
    mov di,160        ;позиция вывода на экран
    cld               ;просмотр вперед - для lodsb и stosw
disp:
    lodsb             ;пересылка байта из ds:si в al
    stosw             ;копирование значения ах
                      ;в es:di (видеобуфер)
    loop disp         ;повтор цикла сх раз
;а теперь выведем количество символов в строке
    mov al,bl
    aam               ;в al две упакованные BCD-цифры
                      ;результата подсчета
    or  ax,3030h      ;преобразование результата в код ASCII
    mov cnt,ah
    mov cnt+1,al
    OutStr  mes       ;вывод строки mes
    Exit              ;макрос выхода
main   endp           ;конец главной процедуры

CountSymbol procnear
;процедура CountSymbol - подсчет символов в строке.
;На входе: si - смещение строки
;На выходе: bl - длина в виде упакованного BCD-числа
    push ax           ;сохранение используемых регистров
    push cx
    cld               ;просмотр вперед
    mov cx,100        ;максимальная длина строки
;блок подсчета символов
go:
    lodsb             ;загрузка символа строки в al
    cmp al,'$'
    je  endstr
    jcxz no_end
    inc Ы             ;приращение счетчика в Ы - количество
                      ;подсчитанных символов в строке
    loop go           ;повтор цикла
endstr:               ;конец строки
    pop cx            ;восстановление регистров из стека
    pop ax
    ret               ;возврат из процедуры
no_end:
;какие-то действия по обработке ситуации
;отсутствия в строке символа $
ret                   ;возврат из процедуры
CountSymbol endp      ;конец процедуры
    end main          ;конец программы
```

Передача аргументов через общую область памяти

Передача аргументов через общую область памяти предполагает, что вызывающая и вызываемая программы условились использовать некоторую область памяти как

общую. Транслятор предоставляет специальное средство для организации такой области памяти. В главе 5 мы разбирали директивы сегментации и их атрибуты. Один из них — атрибут комбинирования сегментов. Наличие этого атрибута указывает компоновщику, как нужно комбинировать сегменты, имеющие одно имя. Значение common означает, что все сегменты, имеющие одинаковое имя в объединяемых модулях, располагаются компоновщиком начиная с одного адреса оперативной памяти. То есть они будут просто перекрываться в памяти и, следовательно, совместно использовать выделенную память.

Недостатком этого способа в реальном режиме работы процессора является отсутствие средств защиты данных от разрушения, так как нельзя проконтролировать соблюдение правил доступа к этим данным.

Рассмотрим листинг 15.4 с примером использования общей области памяти для обмена данными между модулями. На этот раз программа состоит уже из двух независимых модулей, находящихся в разных файлах, и поэтому они представляют собой отдельные единицы трансляции. Функционально эти модули реализуют несложную задачу, которая заключается в том, что вызываемые процедуры формируют строку символов и передают ее через общую область, а вызывающая их процедура main выводит строку на экран.

Листинг 15.4. Передача аргументов через общую область памяти (модуль 1)

```
;prg15_4.asm
include mac.inc         ;подключение файла с макросами
stk segment stack
    db  256 dup (0)
stk ends
common_data segment para common "data"   ;начало общей области памяти
buf db 15 DUP (" ")     ;буфер для хранения строки
temp    dw  0
common_data ends
    extrn   PutChar:far,PutCharEnd:far
code    segment         ;начало сегмента кода
    assume  cs:code,es:common_data
main    proc
    mov ax,common_data
    mov es,ax
;вызов внешних процедур
    call PutChar
    call PutCharEnd
    pushes
    pop ds
    OutStr  buf
    Exit                ;стандартный выход
main    endp            ;конец главной процедуры
code    ends
    end main
```

Вызываемые процедуры находятся в другом модуле (листинг 15.5). Обратите внимание на то, что совсем не обязательно, чтобы данные в сегментах common имели одинаковые имена. Главное, и за этим нужно следить с особой тщательностью, — структура общих сегментов. Она должна быть абсолютно идентичной во всех модулях программы, обменивающихся данными через общую память.

Листинг 15.5. Передача аргументов через общую область памяти (модуль 2)

```
;prg15_5.asm
include mac.inc         ;подключение файла с макросами
```

Листинг 15.5 *(продолжение)*

```
stk segment stack
    db 256 dup (0)
stk ends

pdata   segment para public "data"
mes db  "Общий сегмент",0ah,0dh,'$'
tempi   db  ?
temp2   dd  ?
temp3   dq  ?
pdata   ends
    public  PutChar,PutCharEnd
common_data segment para common "data"   ;начало общей области памяти
buffer  db  15 DUP (" ")    ;буфер для формирования строки
tmpSI   dw  0
common_data ends
code    segment             ;начало сегмента кода
    assume  cs:code,es:common_data,ds:pdata
PutChar proc far            ;объявление процедуры
    cld
    mov si ,0
    irpc ch,<работает!>
    mov buffer[si],'&ch'
    inc si
    endm
    mov tmpSI,si
    ret                     ;возврат из процедуры
PutChar endp                ;конец процедуры
PutCharEnd procfar
    mov si,tmpSI
    mov buffer[si],'$'
    ret
PutCharEnd endp
code    ends
end
```

Так как в данном примере программа состоит уже из двух модулей, то возникает естественный вопрос: как собрать ее в один исполняемый модуль? Можно предложить следующую последовательность шагов.

1. Выполнить трансляцию файла prg15_4.asm и получить объектный модуль prg15_4.obj:

   ```
   tasm /zi prg15_4.asm,,,
   ```

2. Выполнить трансляцию файла prg15_5.asm и получить объектный модуль prg15_5.obj:

   ```
   tasm /zi prg15_5.asm,,,
   ```

3. Скомпоновать программу утилитой TLINK командной строкой вида:

   ```
   tlink /v prg15_4.obj + prg15_4.obj
   ```

В итоге будет создан исполняемый файл prgl5_4.exe. Интересно исследовать его с использованием отладчика. После загрузки prgl5_4.exe в отладчик в окне Module появится только исходный текст программы из файла prg15_4.asm. И лишь после вызова процедуры по команде CALL (нажатием клавиши F7) в окно будет загружен текст вызванной процедуры.

Передача аргументов через стек

Передача аргументов через стек при вызове процедур используется наиболее часто. Суть этого способа заключается в том, что вызывающая процедура самостоя-

тельно заносит в стек передаваемые данные, после чего обращается к вызываемой процедуре. В главе 10 мы рассматривали процессы, происходящие при передаче управления процедуре и возврате из нее. При этом мы обсуждали содержимое стека до и после передачи управления процедуре (см. рис. 10.4 и 10.5). Как следует из этих рисунков, при передаче управления процедуре процессор автоматически записывает на вершину стека два (для процедур типа near) или четыре (для процедур типа far) байта. Вы помните, что эти байты являются адресом возврата в вызывающую программу. Если перед передачей управления процедуре командой CALL в стек были записаны переданные процедуре данные или указатели на них, то они окажутся под адресом возврата.

При рассмотрении архитектуры процессора мы выяснили, что стек обслуживается тремя регистрами: SS, SP и BP. Процессор автоматически работает с регистрами SS и SP в предположении, что они всегда указывают на вершину стека. По этой причине их содержимое изменять не рекомендуется. Для произвольного доступа к данным в стеке архитектура процессора имеет специальный регистр EBP\BP (Base Point — указатель базы). Так же как и для регистра ESP\SP, обращение к EBP\BP автоматически предполагает работу с сегментом стека. Перед использованием этого регистра для доступа к данным стека содержимое стека необходимо правильно инициализировать, что предполагает формирование в нем адреса, который бы указывал непосредственно на переданные данные. Для этого в начало процедуры рекомендуется включить дополнительный фрагмент кода. Он имеет свое название — *пролог* процедуры. Типичный фрагмент программы, содержащий вызов процедуры с передачей аргументов через стек, может выглядеть так:

```
    masm
    model   small
...
    proc_1  proc near    ;"близкая" процедура (near) с n аргументами
    ;начало пролога
        push bp
        mov bp,sp
    ;конец пролога
        mov ax,[bp+4]    ;доступ к аргументу arg_n для near-процедуры
        mov ax,[bp+6]    ;доступ к аргументу arg_{n-1}
    ...                  ;команды процедуры
    ;подготовка к выходу из процедуры
    ;начало эпилога
        mov sp,bp        ;восстановление sp
        pop bp           ;восстановление значения старого bp
                         ;до входа в процедуру
        ret              ;возврат в вызывающую программу
    ;конец эпилога
    proc_1  endp

    .code
    main proc
        mov ax,@data
        mov ds,ax
    ...
    push arg_1           ;запись в стек 1-го аргумента
    push arg_2           :запись в стек 2-го аргумента
    ...
    push arg_n           ;запись в стек n-го аргумента
    call proc_1          ;вызов процедуры proc_1
    ;действия по очистке стека после возврата из процедуры
```

```
<32>      ;..
<33>      ml:
<34>          _exit
<35>      main endp
<36>      end main
```

Код пролога состоит всего из двух команд. Первая команда push bp сохраняет содержимое BP в стеке с тем, чтобы исключить порчу находящегося в нем значения в вызываемой процедуре. Вторая команда пролога mov bp,sp настраивает BP на вершину стека. После этого можно не волноваться о том, что содержимое SP перестанет быть актуальным, и осуществлять прямой доступ к содержимому стека. Что обычно и делается. Для доступа к arg_n достаточно сместиться от содержимого BP на 4, для arg_{n-1} — на 6 и т. д. Но эти смещения подходят только для процедур типа near. Для процедур типа far эти значения необходимо скорректировать еще на 2, так как при вызове процедуры дальнего типа в стек записывается полный адрес — содержимое CS и IP. Поэтому для доступа к arg_n в строке 9 команда будет выглядеть так: mov ax,[bp+6], а для arg_{n-1}, соответственно, mov ax,[bp+8] и т. д.

Конец процедуры также должен быть оформлен особым образом, обеспечивая корректный возврат из процедуры. Фрагмент кода, выполняющего такие действия, имеет свое название — *эпилог* процедуры. Код эпилога должен восстановить контекст программы в точке вызова процедуры из вызывающей программы. При этом, в частности, нужно откорректировать содержимое стека, убрав из него ставшие ненужными аргументы, передававшиеся в процедуру. Это можно сделать несколькими способами.

- Можно использовать последовательность из *n* команд pop xx. Лучше всего это делать в вызывающей программе сразу после возврата управления из процедуры.
- Можно откорректировать регистр указателя стека SP на величину 2 • *n,* например, командой add sp,NN, где NN = 2 • *n,* а *n* — количество аргументов. Это также лучше делать после возврата управления вызывающей процедуре.
- Можно использовать машинную команду RET n в качестве последней исполняемой команды в процедуре. Здесь n — количество байтов, на которое нужно увеличить содержимое регистра ESP\SP после того, как со стека будут сняты составляющие адреса возврата. Видно, что этот способ аналогичен предыдущему, но выполняется процессором автоматически.

В каком виде можно передавать аргументы в процедуру? Ранее упоминалось, что передаваться могут либо данные, либо их адреса (указатели на данные). В языке высокого уровня это называется передачей по значению и по адресу, соответственно.

Наиболее простой способ передачи аргументов в процедуру — передача *по значению.* Этот способ предполагает, что передаются сами данные, то есть их значения. Вызываемая программа получает значение аргумента через регистр или через стек. Естественно, что при передаче переменных через регистр или стек на их размер накладываются ограничения, связанные с размерностью используемых регистров или стека. Другой важный момент заключается в том, что при передаче аргументов по значению в вызываемой процедуре обрабатываются их копии. Поэтому значения переменных в вызывающей процедуре не изменяются.

Способ передачи аргументов *по адресу* предполагает, что вызываемая процедура получает не сами данные, а их адреса. В процедуре нужно извлечь эти адреса тем же методом, как это делалось для данных, и загрузить их в соответствующие регистры. После этого, используя адреса в регистрах, следует выполнить необходимые операции над самими данными. В отличие от передачи данных по значению, при передаче данных по адресу в вызываемой процедуре обрабатывается не копия, а оригинал передаваемых данных. Поэтому при изменении данных в вызываемой процедуре они автоматически изменяются и в вызывающей программе, так как изменения касаются одной области памяти.

Использование директив EXTRN и PUBLIC

Директивы EXTRN и PUBLIC мы уже упоминали, когда рассматривали варианты взаимного расположения вызывающей программы и вызываемой процедуры. Ко всему сказанному добавим, что директивы EXTRN и PUBLIC также можно использовать для обмена информацией между модулями. Назначение и форматы этих директив остаются теми же, поэтому сейчас опишем только порядок их использования для обмена данными. Возможны несколько вариантов их применения:

- оба модуля используют сегмент данных вызывающей программы;
- у каждого из модулей есть собственный сегмент данных;
- модули используют атрибут комбинирования (объединения) сегментов private в директиве сегментации SEGMENT.

Рассмотрим эти варианты на примере программы, которая определяет в сегменте данных две символьные переменные и вызывает процедуру, выводящую эти символы на экран.

В первом варианте, в котором два модуля используют только сегмент данных вызывающей программы, не требуется переопределения сегмента данных в вызываемой процедуре. В листинге 15.6 в вызывающей программе определены две переменные, вывод на экран которых осуществляет вызываемая программа (листинг 15.7).

Листинг 15.6. Первый вариант использования директив EXTRN и PUBLIC (модуль 1)

```
;prg15_6..asm
;Вызывающий модуль
include  mac.inc
extrn    my_proc2:far
public   per1,per2
stk segment stack
    db  256 dup (0)
stk  ends
data     segment
perl    db  "1"
per2     db  "2"
data     ends
code     segment
main      proc far
assume  cs:code,ds:data,ss:stk
    mov ax,data
    mov ds,ax
    call my_proc2
    exit
```

продолжение

Листинг 15.6 *(продолжение)*

```
main     endp
code     ends
end main
```

Листинг 15.7. Первый вариант использования директив extrn и public (модуль 2)

```
;prg15_7.asm
;Вызываемый модуль
include mac.inc
extrn   per1:byte,per2:byte
public  my_proc2
code    segment
my_proc2    procfar
assume  cs:code
;вывод символов на экран
    mov dl,per1
    OutChar
    mov dl,per2
    OutChar
    ret
my_proc2    endp
code    ends
end
```

Сборка программы из двух модулей для этого и следующих вариантов осуществляется так же, как было показано в листингах 15.4 и 15.5.

Во втором варианте у каждого из модулей есть свой собственный сегмент данных. В этом случае для доступа к разделяемым переменным из другого модуля требуется переопределение сегмента данных в вызываемой процедуре (строки 17–19 и 23-24 листинга 15.8).

Листинг 15.8. Второй вариант использования директив extrn и public

```
;prg15_8.asm
;Вызывающий модуль - тот же, что и для предыдущего варианта.
<1> ;Вызываемый модуль
<2> include mac.inc
<3> extrn   per1:byte,per2:byte
<4> public  my_proc2
<5> data segment
<6> per0 db "0"
<7> data ends
<8> code segment
<9> my_proc2 proc   far
<10>    assume  cs:code,ds:data
<11>    ;вывод символов на экран
<12>        mov ax,data
<13>        mov ds,ax
<14>        mov dl,per0
<15>        OutChar
<16>        pushds          ;сохранили ds
<17>        mov ax,seg per1 ;сегментный адрес per1 в ds
<18>        mov ds,ax
<19>        mov dl,per1
<20>        OutChar         ;вывод per1
<21>        mov dl,per2
<22>        OutChar         ;вывод per2
<23>        pop ds          ;восстановили ds
<24>        mov dl,per0
<25>        OutChar         ;и еще раз per0
<26>        ret
```

```
my_proc2 endp
code ends
end
```

Рассмотрим улучшенный второй вариант программы (листинг 15.9). В предыдущем случае мы использовали для адресации данных в разных сегментах данных один регистр DS, а теперь для доступа к разделяемым переменным из другого модуля задействуем один из дополнительных сегментных регистров данных, к примеру ES. Заметьте, что обращение к данным другого сегмента осуществляется с помощью префикса замены сегмента (строки 18 и 20).

Листинг 15.9. Улучшенный второй вариант использования директив extrn и public

```
;prg15_9.asm
;Вызывающий модуль - тот же, что и для предыдущего варианта.
<1>   ;Вызываемый модуль
<2>   include iomac.inc
<3>   extrn   per1:byte,per2:byte
<4>   public  my_proc2
<5>   data segment
<6>   per0 db "0"
<7>   data ends
<8>   code segment
<9>   my_proc2 proc   far
<10>  assume  cs:code,ds:data
<11>  ;вывод символов на экран
<12>      mov ax,data
<13>      mov ds,ax
<14>      mov dl,per0
<15>      OutChar
<16>      mov ax,seg per1
<17>      mov es,ax
<18>      mov dl,es:per1
<19>      OutChar
<20>      mov dl,es:per2
<21>      OutChar
<22>      mov dl,per0
<23>      OutChar
<24>      ret
<25>  my_proc2 endp
<26>  codeends
<27>  end
```

Третий вариант предполагает использование атрибута комбинирования (объединения) сегментов public в директиве сегментации SEGMENT для сегментов данных модулей (листинги 15.10и15.11).Это значение атрибута комбинирования заставляет компоновщик объединить последовательно сегменты с одинаковыми именами. Все адреса и смещения будут вычисляться относительно начала этого нового сегмента. Тогда не понадобится производить дополнительной настройки сегментных регистров (как было в двух предыдущих случаях).

Листинг 15.10. Третий вариант использования директив extrn и public (модуль 1)

```
;prg15_10.asm
;Вызывающий модуль
include mac.inc
extrn   my_proc2:far,per0:byte
public  per1,per2
stk segment stack
    db  256 dup (0)
stk ends
```

продолжение ↗

Листинг 15.10 *(продолжение)*

```
data     segment para public "data"
perl    db  "1"
per2     db  "2"
data     ends
code     segment
main     procfar
assume  cs:code,ds:data,ss:stk
    mov ax,data
    mov ds,ax
    mov dl,per0
    OutChar
call     my_proc2
    exit
main     endp
code     ends
end main
```

Листинг 15.11. Третий вариант использования директив extrn и public (модуль 2)

```
;prg15_11.asm
;Вызываемый модуль
include mac.inc
extrn   per1:byte,per2:byte
public my_proc2,per0
data    segment para public "data"
per0     db  "0"
data     ends
code    segment
my_proc2    proc far
assume  cs:code,ds:data
;ds загружать не надо, так как компоновщик его присоединит
;к сегменту данных первого модуля
;вывод символов на экран
    mov dl,per0
    OutChar
    mov dl,per1
    OutChar
    mov dl,per2
    OutChar
    mov dl,per0
    OutChar
    ret
my_proc2    endp
code    ends
    end
```

Возврат результата из процедуры

В отличие от языков высокого уровня, в языке ассемблера нет отдельных понятий для процедуры и функции. Организация возврата результата из процедуры полностью ложится на программиста. Если исходить из того, что получение результата — частный случай передачи аргументов, то программисту доступны три варианта возврата значений из процедуры.

- С использованием регистров. Ограничения здесь те же, что и при передаче данных, — это небольшое количество доступных регистров и их фиксированный размер. Функции DOS используют именно этот способ. Из рассматриваемых здесь трех вариантов данный способ является наиболее быстрым, поэтому его

есть смысл задействовать для организации критичных по времени вызова процедур с малым количеством аргументов.

- С использованием общей области памяти. Этот способ удобен при возврате большого количества данных, но требует внимательности в определении областей данных и подробного документирования, чтобы устранить неоднозначность при трактовке содержимого общих участков памяти.
- С использованием стека. Здесь, подобно передаче аргументов через стек, также требуется регистр BP. При этом возможны следующие варианты:
  - использование для возвращаемых аргументов тех же ячеек в стеке, которые применялись для передачи аргументов в процедуру, то есть предполагается замещение ставших ненужными входных аргументов выходными данными;
  - предварительное помещение в стек наряду с передаваемыми аргументами фиктивных аргументов с целью резервирования места для возвращаемого значения (в этом варианте процедура, конечно же, не должна пытаться очистить стек командой RET, эту операцию придется делать в вызывающей программе, например, командой POP).

В ходе приведенного обсуждения мы выяснили, что ассемблер не накладывает никаких ограничений на организацию процесса передачи данных и возврата значений между двумя процедурами, а в более общем случае — и между модулями, представляющими отдельные файлы. Наиболее быстрый способ такого обмена — использование регистров. Но часто требуется связывать между собой не только программы, написанные на ассемблере, но и программы на разных языках. В этом случае универсальным является обмен данными через стек. Далее мы рассмотрим, как решается проблема связи программных модулей, написанных на языке ассемблера и языках высокого уровня.

Директива INVOKE

Для более удобного вызова процедур с параметрами, передающимися через стек, MASM предоставляет специальное средство в виде директивы INVOKE:

```
INVOKE имя_процедуры [,аргументы]
```

Основная ее задача — сформировать код, который, во-первых, размещает *аргументы* в стеке, во-вторых, вызывает процедуру и, в третьих, чистит стек после завершения работы процедуры. Например, традиционный способ вызова процедуры выглядит так:

```
    push    параметр_n
...
    push    параметр_2
    push    параметр_1
    call    имя_процедуры
```

С использованием INVOKE тот же самый вызов будет выглядеть следующим образом:

```
INVOKE имя_процедуры, параметр_1, параметр_2 ... параметр_n
```

Для INVOKE аргумент **имя_процедуры** не должен быть опережающей ссылкой на адрес. Чтобы исключить подобные ситуации, существует «парная» для INVOKE директива PROTO:

```
имя_процедуры PROTO [расстояние] [язык] [, [параметр]:тип]...
```

Эта директива информирует ассемблер о количестве и типах аргументов, которые принимает процедура. Использование данной директивы позволяет ассемблеру выполнять проверку типов. Обычно все директивы PROTO для процедур собираются в начале исходного текста программы либо в отдельном включаемом файле. Директива PROTO принимает три типа аргументов — расстояние, язык, параметры процедуры с указанием их типов.

- Аргумент расстояние (NEAR, FAR, NEAR16, NEAR32, FAR16 или FAR32) влияет на размер адреса, формируемого ассемблером для вызова процедуры. По умолчанию значение этого параметра определяется исходя из текущей модели памяти и типа процессора.
- Аргумент язык для определения стиля и соглашения по вызову процедуры в качестве значения принимает имя языка (табл. 15.1).
- Аргумент параметр представляет собой последовательность перечисленных через запятую параметров процедуры. Исходя из этой информации, при вызове процедуры ассемблер преобразует последовательность параметров в последовательность команд PUSH с формированием соответствующих адресов параметров процедуры в стеке.
- Аргумент тип — один из допустимых ассемблером простых типов данных. В качестве типа может быть указано слово VARARG. Оно предназначено для определения процедур с переменным числом аргументов. Тип VARARG указывается с последним параметром, заданным в директиве PROTO. Тип VARARG можно использовать, если аргументом язык является С, SYSCALL или STDCALL.

Таблица 15.1. Передача аргументов в языках высокого уровня

Операнд «язык»	Язык аргументов	Направление передачи стека	Процедура очистки
NOLANGUAGE	Ассемблер	Слева направо	Вызываемая
BASIC	Basic	Слева направо	Вызываемая
PROLOG	Prolog	Справа налево	Вызывающая
FORTRAN	Fortran	Слева направо	Вызываемая
C	C	Справа налево	Вызывающая
C++ (CPP)	C++	Справа налево	Вызывающая
PASCAL	Pascal	Слева направо	Вызываемая
STDCALL	—	Справа налево	Вызываемая
SYSCALL	C++	Справа налево	Вызывающая

Приведем типовую последовательность действий в программе для вызова процедуры в MASM:

```
...
;задание прототипа процедуры prod
procl   PROTO   stdcall :dword, :dword
...
;вызов процедуры
```

```
invoke  prod, arg1, arg2
...
```

Следует отметить разницу в описании процедур в MASM и TASM. Общий вид представления процедуры в MASM похож на стиль TASM, но имеет отличия в синтаксисе и содержании:

```
имя_процедуры PROC [расстояние] [язык] [видимость]
    [<аргументы пролога\эпилога>]
    [USES список_регистров] [,аргумент [:тип]]...
    [LOCAL список_переменных]
...
    тело процедуры
...
[RET [количество_байтов]]
имя_процедуры ENDP
```

Параметры расстояние, язык и видимость имеют сходные с описанием процедуры в TASM значения. Параметр расстояние показывает локализацию места, из которого может быть вызвана процедура, с помощью одного из ключевых слов: NEAR, FAR, NEAR16, NEAR32, FAR16 или FAR32. Параметр язык — один из допустимых типов языков (см. табл. 15.1). Параметр видимость определяет доступность процедуры для других модулей и имеет одно из значений PRIVATE (по умолчанию), PUBLIC или EXPORT. Этот параметр можно изменить директивой OPTION PROC.

Последней командой процедуры обычно является RET. В качестве ее операнда можно указать число байтов, которые процедура должна удалить из стека при возврате управления в точку вызова.

Ассемблер автоматически генерирует код пролога и эпилога для правильной передачи аргументов в процедуру через стек и очистки стека при возврате из нее. Код пролога и эпилога можно несколько изменить применением директив OPTION PROLOGUE и OPTION EPILOGUE.

Локальные переменные объявляются в процедуре директивой LOCAL (перед любыми командами). В процедуре может быть несколько директив LOCAL.

Аргументы пролога/эпилога — аргументы, определяющие особенности пролога и эпилога процедуры. Аргументы по умолчанию — PROLOGUE и EPILOGUE. Другие значения:

- FORCEFRAME — генерация сегмента стека;
- LOADDS — сохранение регистра DS в прологе процедуры и восстановление его в эпилоге.

Если аргументов несколько, то они разделяются запятыми.

При вызове процедуры можно указать регистры общего назначения, которые нужно сохранить в стеке и назначить символические имена адресам в стеке, которые логически являются параметрами процедуры. Все это делает директива USES. Имена перечисляемых регистров разделяются пробелами, а имена переменных — запятыми. Для переменных может указываться тип. В качестве значения типа может быть либо имя простого типа (например, DWORD), либо VARARG. Служебное слово VARARG позволяет задать переменное число параметров. При его использовании оно должно быть последним в списке параметров процедуры. Служебное слово VARARG указывается только, если параметр язык равен C, SYSCALL или STDCALL. По умолчанию параметр тип равен WORD для 16-разрядного сегмента и DWORD — для 32-разрядного сегмента.

Связь ассемблера с языками высокого уровня

На протяжении всего учебника мы неоднократно подчеркивали сильные и слабые стороны языка ассемблера как языка программирования. Писать на нем достаточно объемные программы утомительно. И всегда ли это нужно? Конечно, не всегда. Если программа не предназначена для решения каких-то системных задач, требующих максимально эффективного использования ресурсов компьютера, если к ней не предъявляются сверхжесткие требования по размеру и времени работы, если вы не «фанат» ассемблера — то, на мой взгляд, следует подумать о выборе одного из языков высокого уровня. Существует и третий, компромиссный путь — комбинирование программ на языке высокого уровня с кодом на ассемблере. Такой способ обычно используют в том случае, если в вашей программе есть фрагменты, которые либо вообще невозможно реализовать без ассемблера, либо ассемблер может значительно повысить эффективность работы программы.

Большинство компиляторов учитывают возможность комбинирования их «родного» кода с ассемблером. Как именно? Это зависит от конкретного компилятора языка высокого уровня. Учитывая, что большинство программистов работают или, по крайней мере, владеют основами программирования на языках C/C++ и Pascal, дальнейшее обсуждение будет касаться именно этих языков. В настоящее время существует несколько их основных реализаций, поддерживаемых разными фирмами-производителями. В этих реализациях имеются, в основном, одинаковые механизмы связи с языком ассемблера. Единственное, что остается сделать при реализации конкретной задачи на конкретном компиляторе, — уточнить в документации на язык нужные параметры связи и, возможно, особенности организации связи с кодом на ассемблере. Невозможно дать универсальные рекомендации по этому вопросу и остается лишь сосредоточиться на отображении наиболее принципиальных моментов связи программ на языках Pascal и C/C++ с ассемблером, актуальных для большинства реализаций этих языков.

Вначале мы отметим общие моменты, актуальные как для C/C++, так и для Pascal. Затем на примерах конкретных программ мы обсудим моменты, специфичные для каждого из этих языков.

Существуют следующие формы комбинирования программ на языках высокого уровня с ассемблером.

- Использование операторов типа inline и ассемблерных вставок в виде встраиваемого ассемблерного кода. Эта форма в значительной степени зависит от синтаксиса языка высокого уровня и конкретного компилятора. Она предполагает, что ассемблерные коды в виде команд ассемблера или прямо в машинных командах вставляются в текст программы на языке высокого уровня. Компилятор языка распознает их как команды ассемблера (машинные коды) и без изменений включает в формируемый им объектный код. Эта форма удобна, если надо вставить небольшой фрагмент.
- Использование внешних процедур и функций. Это более универсальная форма комбинирования. У нее есть ряд преимуществ:

- написание и отладку программ можно производить независимо;
- написанные подпрограммы можно использовать в других проектах;
- облегчаются модификация и сопровождение подпрограмм в течение жизненного цикла проекта (к примеру, если ваша процедура на ассемблере производит работу с некоторым внешним устройством, то при смене устройства вам нет необходимости перекраивать весь проект — достаточно заменить только работающую с ним процедуру, оставив неизменным интерфейс программы на языке высокого уровня с ассемблером).

Встраиваемый ассемблерный код

Большинство компиляторов языков высокого уровня поддерживают возможность вставки ассемблерного кода в обрабатываемые ими программы. В данном разделе основное внимание уделяется ассемблерным вставкам в программах на C++, так как именно этот язык наиболее часто используется для реализации задач системного программирования.

Оператор inline языка Pascal представляет собой следующую синтаксическую конструкцию:

```
inline(машинные_коды)
```

В скобках указывается строка машинных кодов. Для получения такой строки целесообразно написать нужный фрагмент на ассемблере, скомпилировать исполняемый модуль, а затем запустить отладчик. В окне CPU отладчика вы увидите машинные коды ваших инструкций; их нужно переписать и вставить в скобки оператора inline.

Встраивание ассемблерного кода в программы C++ производится директивой _asm. Причем этой директивой возможна вставка как одной команды ассемблера, так и группы команд. Синтаксис:

```
__asm   команда_ассемблера  [;комментарий]
__asm {
команда_ассемблера  [;комментарий]
...
команда_ассемблера  [//комментарий или /* комментарий */]
}
```

Планируя использование встроенного ассемблера, важно хорошо представлять себе его возможности и ограничения.

Можно:

- не передавать параметры, как в случае с внешней ассемблерной процедурой;
- иметь непосредственный доступ к командам и регистрам процессора;
- ссылаться на метки и переменные вне текущего блока, находящиеся в пределах видимости ассемблерной вставки;
- вызывать функции вне пределов ассемблерной вставки, причем эти функции должны быть ранее объявлены в программе (на уровне прототипа).
- использовать описание констант как в стиле ассемблера, так и C++;
- использовать операторы PTR, LENGTH, SIZE, TYPE и директивы EVEN и ALIGN.

Нельзя:

- использовать директивы определения данных простых (DB и DD) и сложных типов (STRUC, RECORD), то есть каким-либо образом определять данные любого типа;
- описывать функции в пределах ассемблерной вставки;
- использовать в командах большинство операторов ассемблера типа OFFSET, SEG, SHR, SHL (вместо OFFSET можно использовать LEA);
- использовать любые директивы макроопределений;
- обращаться к полям структур и объединений.

Внешний ассемблерный код

Так как вариант с использованием операторов inline и ассемблерных вставок обладает довольно большими ограничениями, он не может быть признан универсальным, и для реализации сложных задач остается организация связи программ на языках высокого уровня с ассемблерным кодом через внешние процедуры и функции. Возможны два вида такой связи — программа на языке высокого уровня вызывает процедуру на ассемблере и наоборот. В данной главе ограничимся рассмотрением связи только в одну сторону, когда программа на языке высокого уровня вызывает процедуру на ассемблере. Это наиболее часто используемый вид связи.

Вспомним (см. главу 10) синтаксис директивы PROC компилятора TASM:

```
имя_процедуры PROC [[модификатор_языка]язык] [расстояние]
```

Один из операндов — язык. Он служит для того, чтобы компилятор мог правильно организовать интерфейс (связь) между процедурой на ассемблере и программой на языке высокого уровня. Необходимость такого указания возникает вследствие того, что способы передачи аргументов при вызове процедур различны для разных языков высокого уровня.

TASM поддерживает несколько значений операнда язык. Ранее в табл. 15.1 для некоторых из этих значений были приведены характерные особенности передачи аргументов и соглашения о том, какая процедура очищает стек — вызывающая или вызываемая. Под *направлением передачи аргументов* понимается порядок, в котором аргументы включаются в стек, по сравнению с порядком их следования в вызове процедуры. Для языка Pascal характерен прямой порядок включения аргументов в стек: первым в стек записывается первый передаваемый аргумент из оператора вызова процедуры, вторым — второй аргумент и т. д. На вершине стека после записи всех передаваемых аргументов оказывается последний аргумент. Для языков C/C++, наоборот, характерен обратный порядок передачи аргументов. В соответствии с ним в стек сначала включается последний аргумент из оператора вызова процедуры (или функции), затем предпоследний и т. д. В конечном итоге на вершине стека оказывается первый аргумент. Это делает возможной передачу переменного количества параметров при вызове функций в языках C/C++. Наверху стека оказывается первый параметр функции, значение которого — предоставление информации о количестве аргументов в данном вызове функции. Что же касается очистки стека, то понятно, что должны быть определенные договорен-

ности об этом. В языке Pascal эту операцию всегда совершает вызываемая процедура, в языках C/C++ — вызывающая. При разработке программы с использованием только одного языка высокого уровня об этом задумываться не имеет смысла, но если мы собираемся связывать несколько «разноязыких» модулей, то эти соглашения нужно иметь в виду.

Вспомните действия, которые мы проделывали для того, чтобы настроиться на аргументы в стеке. Теперь, после указания языка, с программой на котором должна осуществляться связь, все действия по настройке стека будут производиться компилятором. При этом в текст процедуры он включит дополнительные команды входа в процедуру (пролог) и выхода из нее (эпилог), причем код эпилога может повторяться несколько раз — перед каждой командой RET. Для значения NOLANGUAGE и по умолчанию коды пролога и эпилога не создаются.

Для пояснения последних замечаний рассмотрим на конкретных примерах организацию связей между модулями на ассемблере и модулями наиболее популярных языков высокого уровня — C и Pascal. Не стоит воспринимать нижеследующие примеры как образец оптимальности — они лишь в учебных целях призваны проиллюстрировать принципы организации межъязыковых связей.

Pascal и ассемблер

Организацию связи языков Pascal и ассемблер рассмотрим на следующем примере: разработаем программу на языке Pascal, которая выводит символ заданное количество раз начиная с определенной позиции на экране (листинги 15.12 и 15.13). Все числовые аргументы определяются в программе на Pascal. Вывод символа осуществляет процедура ассемблера. Очевидно, что основная проблема в этой задаче — организация взаимодействия модулей на Pascal и ассемблере.

Листинг 15.12. Взаимодействие Pascal—ассемблер (модуль на Pascal)

```
{prg15_12.pas}
{Программа, вызывающая процедуру на ассемблере}
program ray_pas;
{$D+} {включение полной информации для отладчика}
uses crt;
procedure asmproc(ch:char;x,y,kol:integer); external;
{процедура asmproc обьявлена как внешняя}
{$L c:\bp\work\prg15_12.obj}
BEGIN
    clrscr; {очистка экрана}
    asmproc('a',1,4,5);
    asmproc('s',9,2,7);
END.
```

Листинг 15.13. Взаимодействие Pascal—ассемблер (модуль на ассемблере)

```
;prg15_12.asm
;Процедура на ассемблере, которую вызывает
;программа на Pascal.
;Для вывода на экран используются службы BIOS:
;02h - позиционирование курсора.
;09h - вывод символа заданное количество раз.
MASM
MODEL   small
STACK   256
.code
```

продолжение ⇨

Листинг 15.13 *(продолжение)*

```
<11>    asmproc proc near
<12>    PUBLIC  asmproc       ;объявлена как внешняя
<13>        pushbp            ;пролог
<14>        mov bp,sp
<15>        mov dh,[bp+6]     ;номер строки для вывода
<16>                          ;символа у - в dh
<17>    mov dl,[bp+8]         ;номер столбца для вывода
<18>;символа х - в dl
<19>    mov ah,02h            ;номер службы BIOS
<20>    int 10h               ;вызов прерывания BIOS
<21>;вызов функции 09h прерывания BIOS 10h:
<22>;вывод символа из a\ на экран
<23>    mov ah,09h            ;номер службы BIOS
<24>    mov al,[bp+10]        ;символ ch в al
<25>    mov bl,07h            ;атрибут символа - в bl
<26>    xor bh,bh
<27>    mov cx,[bp+4]         ;копичество "выводов"
<28>;символа - в cx
<29>    int 10h               ;вызов прерывания BIOS
<30>    pop bp                ;восстановление bp
<31>;очистка стека и возврат из процедуры
<32>    ret 8
<33>asmproc endp            ;конец процедуры
<34>end
```

Процесс организации такой связи состоит из нескольких шагов.

1. Написать процедуру на ассемблере дальнего (far) или ближнего типа (near). Назовем ее для примера asmproc. В программе на языке ассемблера (назовем ее prg15_13.asm), в которую входит процедура asmproc, необходимо объявить имя этой процедуры внешним с помощью директивы PUBLIC:

   ```
   PUBLIC asmproc
   ```

   Для того чтобы процедура на ассемблере при компоновке с программой на Pascal воспринималась компилятором Borland Pascal 7.0 как far или near, недостаточно просто объявить ее таковой в директиве PROC (строка 11 листинга 15.13). Кроме того, вам нужно включить или выключить параметр компилятора, доступный через меню интегрированной среды: Options ▶ Compiler ▶ Force far calls. Установка этого параметра заставляет компилятор генерировать дальние вызовы подпрограмм. Альтернатива данного параметра — ключ {$F+} или {$F-} (соответственно, включено или выключено) в программе. Это — локальные ключи, то есть в исходном тексте программы на Pascal их может быть несколько, и они могут, чередуясь друг с другом, поочередно менять форму генерируемых адресов перехода: для одних подпрограмм — дальние вызовы, для других — ближние.

2. Произвести компиляцию программы prg15_12.asm с целью устранения синтаксических ошибок и получения объектного модуля программы prg15_12.obj:

   ```
   tasm /zi prg15_12,,,
   ```

3. В программе prg15_12.pas на Pascal, которая будет вызывать внешнюю процедуру на ассемблере, следует вставить директиву компилятора {$L \путь\prg_15_12.obj}. Эта директива заставит компилятор в процессе компиляции программы prg15_12.pas загрузить с диска объектный модуль программы prg15_12.obj. В программе prg15_12.pas необходимо объявить процедуру asmproc как внешнюю. В итоге последние два объявления в программе на Pascal будут выглядеть так:

   ```
   {$L my_asm}procedure asmproc(ch:char;kol,x,y:integer); external;
   ```

4. Если вы собираетесь исследовать в отладчике работу программы, то необходимо потребовать, чтобы компилятор включил отладочную информацию в генерируемый им исполняемый модуль. Для этого есть две возможности. Первая заключается в использовании глобального ключа {$D+}. Этот ключ должен быть установлен сразу после заголовка программы на Pascal. Вторая альтернативная возможность заключается в установке параметра компилятора: Options ▸ Compiler ▸ Debug Information.

5. Выполнить компиляцию программы на Pascal. Для компиляции удобно использовать интегрированную среду. Для изучения особенностей связки Pascal — ассемблер удобно прямо в интегрированной среде перейти к работе в отладчике командой Tools ▸ Turbo Debugger (или клавишами Shift+F4). Будет загружен отладчик. Его среда вам хорошо знакома; в данном случае в окне Module вы увидите текст программы на Pascal. Нажимая клавишу F7, вы в пошаговом режиме будете исполнять программу на Pascal. Когда очередь дойдет до вызова процедуры на ассемблере, отладчик откроет окно с текстом программы на ассемблере. Но наш совет вам — не ждать этого момента, так как вы пропустите некоторые интересные вещи. Дело в том, что отладчик скрывает момент перехода из программы на Pascal в процедуру на ассемблере. Поэтому лучше всего исполнять программу при открытом окне CPU отладчика. И тогда вы станете свидетелями тех процессов, которые мы будем обсуждать далее.

Если бы взаимодействие программ ограничивалось только передачей и возвратом управления, то на этом обсуждение можно было бы и закончить. Но дело значительно усложняется, когда требуется передать аргументы (в случае процедуры) или передать аргументы и возвратить результат (в случае функции). Рассмотрим процессы, которые при этом происходят.

Передача аргументов при связи модулей на разных языках всегда производится через стек. Компилятор Pascal генерирует соответствующие команды при обработке вызова процедуры ассемблера. Это как раз те команды, которые отладчик пытался скрыть от нас. Они записывают в стек аргументы и генерируют команду CALL для вызова процедуры ассемблера. Чтобы убедиться в этом, просмотрите на исполняемый код программы в окне CPU отладчика. После обработки вызова процедуры и в момент передачи управления процедуре asmproc содержимое стека будет таким, как показано на рис. 15.1,*а*. Для доступа к этим аргументам можно применять различные методы, наиболее удобный из них — использование регистра BP.

Регистр BP, как уже отмечалось, специально предназначен для организации произвольного доступа к стеку. Когда мы рассматривали связь ассемблерных модулей, то говорили о необходимости добавления в текст вызываемого модуля фрагментов, настраивающих его на передаваемые ему аргументы. При объединении разноязыких модулей также нужно вставлять подобные дополнительные фрагменты кода. Они, кроме всего прочего, позволят учесть особенности конкретного языка. Фрагмент, вставляемый в самое начало вызываемого модуля, называется *прологом* модуля (процедуры). Фрагмент, вставляемый перед командами передачи управления вызывающему модулю, называется *эпилогом* модуля (процедуры). Его назначение — восстановление состояния вычислительной среды на момент вызова данного модуля.

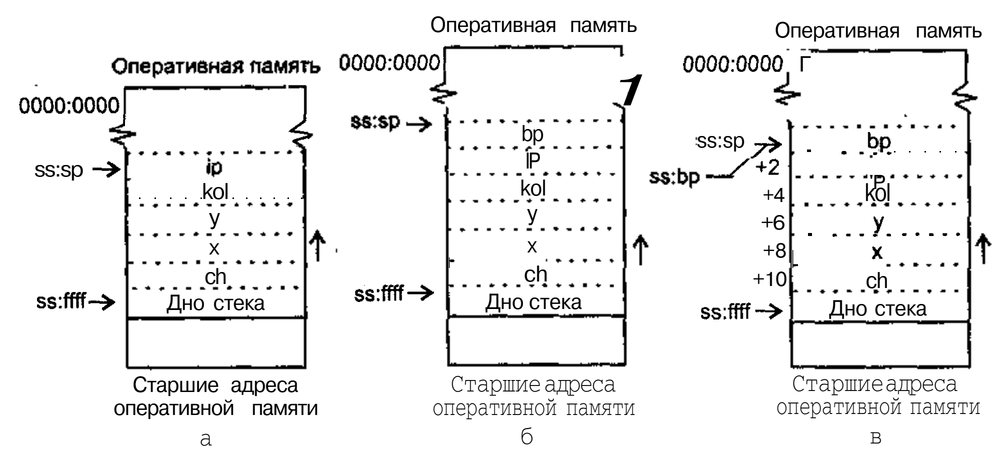


Рис. 15.1. Изменение содержимого стека при передаче управления в связке Pascal—ассемблер

Рассмотрим действия, выполняемые кодами пролога и эпилога при организации связи Pascal—ассемблер.

Действия, выполняемые кодом пролога.

1. Сохранить значение Ьр в стеке. Это делается с целью запоминания контекста вызывающего модуля. Стек при этом будет выглядеть, как показано на рис. 15.1, *б*.
2. Записать содержимое SP в BP. Тем самым BP теперь тоже будет указывать на вершину стека (рис. 15.1, в).

После написания кода пролога все обращения к аргументам в стеке можно организовывать относительно содержимого регистра BP. Из рис. 15.1, *в* видно, что для обращения к верхнему и последующим аргументам в стеке содержимое Ьр необходимо откорректировать. Нетрудно посчитать, что величина корректировки будет различаться для процедур дальнего (far) и ближнего (near) типов. Причина понятна: при вызове процедуры типа near в зависимости от установленного режима адресации (use16 или use32) в стек записывается 2(4) байта в качестве адреса возврата (содержимое ip/eip), а при вызове процедуры типа far в стек записывается 4(8) байта (содержимое IP/EIP и CS)[1].

Таким образом, коды пролога для процедур ближнего и дальнего типа соответственно будут выглядеть следующим образом:

```
asmproc procnear
;пролог для процедуры типа near
    push    bp
    mov bp,sp
;к прологу можно добавить команду
;корректировки bp на 4 с тем, чтобы регистр bp
;указывал на верхний из передаваемых аргументов в стеке
    add bp,4;теперь bp указывает на kol
...
asmproc proc  far
```

[1] Если действует режим адресации use32, то в стек записываются двойные слова. По этой причине запись 16-разрядного регистра cs также производится четырьмя байтами, при этом два старших байта этого значения нулевые.

```
;пролог для процедуры типа far
    push     bp
    raovbp,sp
;к прологу можно добавить команду
;корректировки bp на 6 с тем, чтобы регистр bp
;указывал на верхний из передаваемых аргументов в стеке
    add bp,6;теперь bp указывает на kol
...
```

Далее доступ к переданным в стеке данным осуществляется, как показано в листинге 15.7.

Как видите, все довольно просто. Но если мы вдруг решили изменить тип нашей процедуры ассемблера с far на near или наоборот, то нужно явно изменить и код пролога. Это не совсем удобно. TASM предоставляет выход в виде директивы ARG, которая служит для работы с аргументами процедуры. Синтаксис директивы ARG иллюстрирует рис. 15.2.

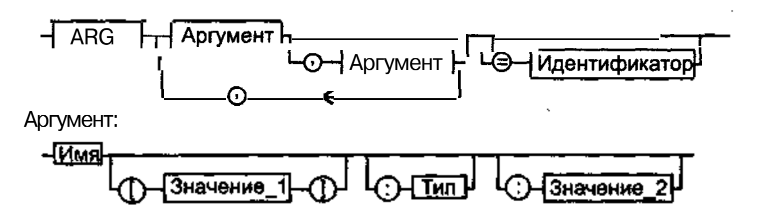


Рис. 15.2. Синтаксис директивы ARG

Несколько слов об обозначениях на рисунке:

- *имя* — идентификатор переменной, который будет использоваться в процедуре на ассемблере для доступа к соответствующей переменной в стеке;
- *тип* — тип данных аргумента (по умолчанию WORD для use16 и DWORD для use32);
- *значение_1* определяет количество аргументов с данным именем. Место в стеке для них будет определено, исходя из расчета:

  значение_1 • значение_2 • размер_типа.

  По умолчанию *значение_1* — 1;
- *значение_2* определяет, сколько элементов данного типа задает данный аргумент. По умолчанию его значение равно 1, но для типа byte *значение_2* = 2, так как стековые команды не работают с отдельными байтами. Хотя, если явно задать *значение_2* = 1, то транслятор действительно будет считать, что в ячейку стека помещен один байт;
- *идентификатор* — имя константы, значение которой присваивает транслятор. Об идентификаторе мы подробно поговорим чуть позже.

Таким образом, директива ARG определяет аргументы, передаваемые в процедуру. Ее применение позволяет обращаться к аргументам по их именам, а не по смещениям относительно содержимого BP. К примеру, если в начале рассматриваемой нами процедуры на ассемблере asmproc задать директиву ARG в виде arg kol:word,y:word, x:word,chr:byte, то к аргументам процедуры можно будет обращаться по их именам, без подсчета смещений. Ассемблер сам выполнит всю необходимую

работу. В этом можно убедиться, запустив программу в отладчике. Обратите внимание: порядок следования аргументов в директиве arg является обратным порядку их следования в описании процедуры (строка procedure asmproc(ch:char;x,y, kol:integer); external; в программе на Pascal). Процедура asmproc с директивой arg представлена в листинге 15.14.

Листинг 15.14. Использование директивы arg

```
{prg15_14.pas}
{Программа на Pascal, вызывающая процедуру на ассемблере, полностью
совпадает с листингом 15.12}
;prg15_14.asm
MASM
MODEL   small
STACK   256
.code
main:
asmproc proc near
;обьявление  аргументов:
arg kol:WORD,y:WORD,x:WORD,chr:BYTE=a_size
PUBLIC  asmproc
    push    bp          ;сохранение указателя базы
    mov bp,sp           ;настройка bp на стек через sp
    mov dh,byte ptr y   ; y в dh
    mov dl,byte ptr x   ; x в dl
    mov ah,02h          ;номер службы BIOS
    int 10h             ;вызов прерывания BIOS
    mov ah,09h          ;номер службы BIOS
    mov al,chr          ;символ - в al
    mov bl,07h          ;маска вывода символа
    xor bh,bh
    mov cx,kol          ;kol в cx
    int 10h             ;вызов прерывания BIOS
    pop bp              ;эпилог
    ret a_size          ;будет ret 8 и выход из процедуры
asmproc endp            ;конец процедуры
end main                ;конец программы
```

После того как решена проблема передачи аргументов в процедуру и выполнены все необходимые действия, возникает очередной вопрос: как правильно возвратить управление? При возврате управления в программу на Pascal нужно помнить, что соглашения этого языка требуют, чтобы вызываемые процедуры самостоятельно очищали за собой стек. Программа на ассемблере также должна удовлетворять этому требованию и заботиться об очистке стека перед своим завершением. Для этого необходимо составить эпилог.

Действия, выполняемые кодом эпилога для связи Pascal—ассемблер.

1. Записать содержимое bp в sp командой mov sp,bp. Это действие восстанавливает в sp значение, которое было на момент входа в процедуру. Необходимость в этом действии возникает в том случае, если в процедуре производилась работа со стеком. В листинге 15.13 такой работы не было, поэтому код эпилога реализует только следующие два действия.
2. Восстановить сохраненный в стеке регистр BP.
3. Удалить из стека переданные процедуре аргументы.

Для удаления из стека аргументов можно использовать различные способы.

И Можно явно скорректировать значение SP, переместив указатель стека на необходимое количество байтов в положительную сторону. Это — не универсальный способ, к тому же он чреват ошибками, особенно при частых модификациях программы.

11 Можно использовать в директиве arg после записи последнего аргумента операнд, состоящий из символа равенства (=) и идентификатора, указанного за ним в следующей синтаксической конструкции:

```
=идентификатор
```

В этом случае TASM при обработке директивы arg подсчитает количество байтов, занятых всеми аргументами, и присвоит их значение идентификатору. В нашем случае директиву arg можно определить так:

```
arg ch:byte;x:word;y:word;kol:word=a_size
```

TASM после обработки данной директивы присвоит имени a_size значение 8 (байт). Это имя впоследствии нужно будет указать в качестве операнда команды ret:

```
ret a_size
```

Есть еще одна возможность организации данных Pascal—ассемблер — использовать операнды директивы MODEL. Вы помните, что она позволяет задать модель памяти и учесть соглашения языков высокого уровня о вызове процедур. Для связи Pascal—ассемблер ее можно задавать в виде

```
MODEL   large,pascal
```

Задание в таком виде директивы MODEL позволяет:

- описать аргументы процедуры непосредственно в директиве proc:

```
asmproc   proc   near   ch:byte,x:word,y:word,kol:word
```

- автоматически сгенерировать код пролога и эпилога в процедуре на ассемблере;
- для доступа к аргументам, объявленным в PROC, использовать их имена (в этом отношении данный вариант является аналогом предыдущего варианта с директивойARG).

Листинг 15.15 демонстрирует, как отражаются особенности данного варианта на тексте процедуры ассемблера. Обратите внимание на то, что пролога уже нет, так как он формируется транслятором автоматически; вместо эпилога обязательно нужно задавать команду RET, только без операндов. Интересно изучить текст листинга 15.16, который получается в результате трансляции листинга 15.15. В нем видны сформированные транслятором коды пролога и эпилога. Кроме того, транслятор заменил команду RET без операндов командой ret 0008, которая, в соответствии с требованиями к взаимодействию с программами на Pascal, удалит из стека аргументы, переданные вызываемой процедуре.

Листинг 15.15. Использование директивы MODEL

```
{prg15_15.pas}
{Программа на Pascal, вызывающая процедуру на ассемблере, полностью
совпадает с листингом 15.12}
;prg15_15.asm
MASM
MODEL   large,pascal
```

продолжение ⇨

Листинг 15.15 *(продолжение)*

```
STACK   256
.code
asmproc proc near chr:BYTE,x:WORD,y:WORD,kol:WORD
PUBLIC  asmproc
    mov dh,byte ptr y     ; y в dh
    mov dl,byte ptr x     ; x в dl
    mov ah,02h            ;номер службы BIOS
    int 10h               ;вызов прерывания BIOS
    mov ah,09h            ;номер службы BIOS
    mov al,chr            ;символ - в al
    mov bl,07h            ;маска вывода символа
    xor bh,bh
    mov cx,kol            ;kol в cx
    int 10h               ;вызов прерывания BIOS
ret
asmproc endp              ;конец процедуры
end                       ;конец программы
```

Листинг 15.16. Результат трансляции листинга 15.15

```
Turbo AssemblerVersion 4.117/04/98 22:30:57Page 1
prg14_82.asm
1 ;prg14_92.asm
2 MASM
3 0000  MODEL   large,pascal
4 0000  STACK   256
5 0000  .code
6 0000
7 0000  asmproc procnear chr:BYTE,x:WORD,y:WORD,kol:WORD
8       PUBLIC  asmproc
1 9     0000 55     PUSH    BP
1 10    0001 8B EC  MOV BP,SP
1 11    0003 8A 76 06   mov dh,byte ptr y    ; y в dh
  12    0006 8A 56 08   mov dl,byte ptr x    ; x в dl
  13    0009 B4 02      mov ah,02h           ;номер службы BIOS
  14    000B CD 10      int 10h              ;вызов прерывания BIOS
  15    000D B4 09      mov ah,09h           ;номер службы BIOS
  16    000F 8A 46 0A   mov al,chr           ;символ - в al
  17    0012 B3 07      mov bl,07h           ;маска вывода символа
  18    0014 32 FF      xor bh,bh
  19    0016 8B 4E 04   mov cx,kol           ;kol в cx
  20    0019 CD 10      int 10h              ;вызов прерывания BIOS
1 21    001B 5D         POP BP
1 22    001C C2 0008RET 00008h
  23    001F asmproc endp                    ;конец процедуры
  24    end                                  ;конец программы
```

Таковы стандартные способы вызова ассемблерных процедур из программ на Pascal и передачи им аргументов. Эти способы будут работать всегда, но, совершенствуясь, компилятор может предоставлять и более удобные средства. Их мы рассматривать не будем, так как, в конечном счете, они сводятся к рассмотренной нами процедуре. Остались открытыми два вопроса.

- Как быть с передачей данных остальных типов Pascal, ведь мы рассмотрели только данные размером в байт и слово?
- Как возвратить значение в программу на Pascal?

Что касается ответа на первый вопрос, то необходимо вспомнить, что в языке Pascal существуют два способа передачи аргументов в процедуру: по ссылке и по значению.

Тип аргументов, передаваемых по *ссылке,* совпадает с типом ассемблера dword и с типом pointer в Pascal. По сути, это указатель из четырех байтов на некоторый объект. Структура указателя обычная: два младших байта — смещение, два старших байта — значение сегментной составляющей адреса. С помощью такого указателя в программу на ассемблере передаются адреса следующих объектов:

- всех аргументов, объявленных при описании в программе на Pascal как var, независимо от их типа;
- аргументов pointer и longint;
- строк string;
- множеств;
- массивов и записей, имеющих размер более четырех байтов.

Аргументы по *значению* передаются следующим образом:

- для типов char и byte — как байт;
- для типа boolean — как байт со значением 0 или 1;
- для перечисляемых типов со значением 0...255 — как байт; более 255 — как два байта;
- для типов integer и word — как два байта (слово);
- для типа real — как шесть байтов (три слова);
- массивы и записи, длина которых не превышает четырех байтов, передаются «как есть».

Заметим, что аргументы таких типов, как single, double, extended и comp, передаются через стек сопроцессора.

Что касается ответа на второй вопрос, то мы выясним его на конкретном примере. Напомню, что мы рассматриваем вызов из программы на Pascal внешней процедуры на ассемблере. Понятно, что вызов ради вызова вряд ли нужен — вызываемая процедура должна иметь возможность вернуть данные в вызывающую программу. Поэтому такую вызываемую процедуру правильнее рассматривать как функцию. В связке Pascal—ассемблер для того, чтобы возвратить результат, процедура на ассемблере должна поместить его значение в строго определенное место (табл. 15.2).

Таблица 15.2. Возврат результата из процедуры на ассемблере в программу на Pascal

Тип возвращаемого значения	Место записи результата
Байт	AL
Слово	AX
Двойное слово	DX:AX (старшее слово:младшее слово)
Указатель	DX:AX (сегмент:смещение)

В листинге 15.17 приведен текст вызывающего модуля на Pascal, а в листинге 15.18 — код вызываемого модуля на ассемблере. Программа на Pascal инициализирует две переменные, value1 и value2, после чего вызывает функцию на ассемблере AddAsm для их сложения. Результат возвращается в программу на Pascal и присваивается переменной rez.

Листинг 15.17. Вызывающая программа на Pascal

```
{prg15_17.pas}
program prg14101;
{внешние объявления}
function  AddAsm:word; external;
{$L prg15_18.obj}
var
value1:word;{здесь как внешние}
value2:word;
rez:word;
begin
value1:=2;
value2:=3;
{вызов функции}
rez:=AddAsm;
writeln("Результат: ",rez);
end.
```

Листинг 15.18. Вызываемая процедура на ассемблере

```
;prg15_18.asm
MASM
MODEL   small
data    segment word public ;сегмент данных
;объявление внешних переменных
extrn   value1:WORD
extrn   value2:WORD
data    ends;конец сегмента данных
.code
assume  ds:data;привязка ds к сегменту
;данных программы на Pascal
main:
AddAsm   procnear
PUBLIC   AddAsm    ;внешняя
    mov cx,ds:value1;value1в cx
    mov dx,ds:value2; value2В dx
    add cx,dx    ;сложение
    mov ax,cx    ;результат в ах, так как - слово
    ret      ;возврат из функции
    AddAsm  endp    ;конец функции
end main;конец программы
```

В последней программе следует обратить внимание еще на одну возможность доступа к разделяемым данным — с помощью сегментов типа PUBLIC (см. главу 5). Совместное использование сегментов данных стало возможным благодаря тому, что компилятор Pascal создает внутреннее представление программы в виде сегментов, как и положено программе, выполняющейся в архитектруре IA-32 на процессоре Intel. Сегмент данных в этом представлении тоже имеет название data, и директива SEGMENT для него со всеми вытекающими последствиями выглядит так:

```
data segment word public
```

Команды ENTER и LEAVE

Учитывая важность проблемы организации межмодульных связей, в систему команд процессора были введены специальные команды ENTER и LEAVE. Их использование позволяет облегчить написание кода пролога и эпилога в процедурах ассемблера, например:

```
;старый код пролога:
    push    bp
    mov bp,sp
;новый код пролога:
    enter   0,0
;...
;старый код эпилога:
    mov  sp,bp
    pop bp
;новый код эпилога:
    leave
```

Транслятор ассемблера предоставляет средства в виде директив, которые еще больше упрощают работу программиста по формированию кодов пролога и эпилога. Одной из них является директива ARG. Применение этой и других директив обсудим на конкретном примере. В главе 16 рассматриваются довольно сложные программы, в которых, в частности, используются обсуждаемые здесь директивы. В листинге 16.7 производится обращение к процедуре WindowProc. При этом ей в стеке передается ряд параметров. Кроме того, в процедуре имеются локальные переменные. Заголовок и конец процедуры выглядят следующим образом:

```
;-----------WindowProc--------------------------------
WindowProcproc
arg@@hwnd:DWORD, @@mes:DWORD, @@wparam:DWORD, @@lparam:DWORD
uses ebx,edi,esi,ebx;эти регистры обязательно должны сохраняться
local@@hdc:DWORD,@@hbrush:DWORD,@@hbit:DWORD
...
exit_wndproc:
ret
WindowProc endp
```

В этом фрагменте новыми средствами ассемблера для нас являются директивы USES и LOCAL.

Директива USES содержит список регистров (см. рис. 10.3 в главе 10). Ее использование заставляет транслятор генерировать код для сохранения в стеке этих регистров при входе в процедуру и их восстановления при выходе из нее.

Директива LOCAL (см. рис. 10.3) позволяет задействовать в процедуре локальные переменные. Эти переменные должны быть перечислены в списке аргументов вместе с их типами. Информацию о количестве и типах переменных транслятор использует для формирования соответствующего программного кода.

Оттранслируем исходный текст программы из листинга 16.7 и откроем для просмотра файл prg16_7.lst. Посмотрим на результат применения этих директив:

```
;-----------WindowProc--------------------------------
0000012D    WindowProc  proc
    arg @@hwnd:DWORD, @@mes:DWORD, @@wparam:DWORD, @@lparam:DWORD
    uses    ebx,edi,esi
    local   @@hdc:DWORD,@@hbrush:DWORD,@@hbit:DWORD
0000012D C8 000C 00 ENTERD  0000Ch,0
00000131  53     PUSH ebx
00000132  57     PUSH edi
00000133  56     PUSH esi
00000134  83 7D 0C 02    cmp @@mes,WM_DESTROY
;...
000002CF      exit_wndproc:
000002CF 5E POP esi
000002D0 5F POP edi
000002D1 5B POP ebx
000002D2 C9 LEAVED
```

```
000002D3 C20010 RET 00010h
000002D6    WindowProc  endp
```

Видно, что транслятор хорошо «поработал» над кодами входа в процедуру и выхода из нее. Директива USES ebx, edi, esi заставляет транслятор генерировать команды PUSH и POP для сохранения-восстановления регистров EBX, EDI и ESI. А какое влияние на формирование кода входа в процедуру и выхода из нее оказывают директивы ARG и LOCAL? Чтобы разобраться с этим, выполним трансляцию трех вариантов программы из листинга 16.7 (глава 16), в каждом из которых закомментируем определенные строки:

- Закомментируем строку с директивой ARG. Фрагмент листинга будет выглядеть следующим образом:

```
;----------WindowProc----------------------------.
0000012DWindowProc  proc
    ;arg@@hwnd:DWORD, @@mes:DWORD, @@wparam:DWORD, @@lparam:DWORD
    uses ebx,edi,esi;эти регистры обязательно должны сохраняться
    local   @@hdc:DWORD,@@hbrush:DWORD,@@hbit:DWORD
0000012D C8 000C 00 ENTERD  0000Ch,0
00000131 53 PUSH ebx
00000132 57 PUSH edi
00000133 56 PUSH esi
00000134 83 3D 00000000 02  cmp @@mes,WM_DESTROY
;...
000002F9exi t_wndproc:
000002F9 5E    POP esi
000002FA 5F    POP edi
000002FB 5B    POP ebx
000002FC C9     LEAVED
000002FD C3     RET 00000h
000002FEWindowProc  endp
```

- Закомментируем строки с директивами LOCAL и USES. Фрагмент листинга будет выглядеть так:

```
;----------WindowProc---------------------------
0000012DWindowProc  proc
    arg @@hwnd:DWORD, @@mes:DWORD, @@wparam:DWORD, @@lparam:DWORD
    ;uses ebx,edi,esi ;эти регистры обязательно должны сохраняться
    ;local @@hdc:DWORD,@@hbrush:DWORD,@@hbit:DWORD
0000012D C8 0000 00 ENTERD  00000h,0
00000131 83 7D0C 02 cmp     @@mes,WM_DESTROY
;...
000002E6exit_wndproc:
000002E6 C9      LEAVED
000002E7 C2 0010 RET 00010h
000002EAWindowProc  endp
```

- Наконец, закомментируем строки с директивами LOCAL и ARG. Фрагмент листинга:

```
;----------WindowProc---------------------------.
0000012DWindowProc  proc
    ;arg @@hwnd:DWORD, @@mes:DWORD, @@wparam:DWORD, @@lparam:DWORD
    uses ebx,edi,esi ;эти регистры обязательно должны сохраняться
    ;local @@hdc:DWORD,@@hbrush:DWORD,@@hbit:DWORD
0000012D 53     PUSH ebx
0000012E 7      PUSH edi
0000012F 56     PUSH esi
00000130 83 3D 00000000 02  cmp @@mes,WM_DESTROY
**Error** prg16_3.asm(209)  PROCBEG(4) Undefinedsymbol:@@mes
*Warning* prg16_3.asm(209)  PROCBEG(4) Argument needs type override
;...
```

```
0000030Fexit_wndproc:
0000030F 5E      POP esi
00000310 5F     POP edi
00000311 5B     POP ebx
00000312  C3     RET 00000h
00000313WindowProc  endp
```

Проанализируем эти фрагменты.

Применение директив ARG и LOCAL приводит к генерации команд ENTER и LEAVE при входе и выходе из процедуры. Поочередное комментирование этих директив показывает, что они влияют только на формирование операнда команды ENTER. При использовании директивы LOCAL он равен числу байтов, необходимых для размещения в стеке локальных переменных. Это так называемый *кадр стека.* Если строку с директивой LOCAL закомментировать, то команда ENTER все равно формируется, но с нулевым значением первого операнда. Это говорит о том, что пролог процедуры создается в любом случае, но директива LOCAL позволяет еще и сформировать кадр стека для хранения локальных переменных процедуры. Соответственно, во всех вариантах генерации команды ENTER в конце процедуры формируется команда LEAVE.

Теперь посмотрим на то, какое влияние оказывает директива ARG на формирование команды RET в процедуре WindowProc. Из анализа четырех приведенных ранее вариантов фрагмента листинга видно, что в некоторых из них транслятор вычисляет суммарный размер аргументов, передаваемых в процедуру, и делает это значение операндом команды RET. Это производится в тех случаях, когда директива ARG не закомментирована. Отсюда следует вывод о прямом влиянии директивы ARG на операнд команды RET. Ненулевое значение операнда команды RET приводит к изменению значения регистра SP\ESP — в нашем случае это означает очистку стека от аргументов, переданных в процедуру.

Отметим, что эти средства можно использовать не только для связи Pascal—ассемблер, но и для организации других межъязыковых связей, в том числе ассемблер—ассемблер.

С и ассемблер

Общие принципы организации связи С—ассемблер напоминают только что рассмотренное соединение Pascal и ассемблера. Поэтому мы коротко обсудим различия на примере конкретных программ. Но прежде отметим, что хотя язык С++ предоставляет дополнительные возможности связи программы с ассемблером, одновременно в нем продолжает поддерживаться традиционная организация связи. Поэтому мы рассмотрим связь с ассемблером в стиле С как стандартную. При необходимости читатель, зная основы подобной связи, без труда разберется с нюансами дополнительных возможностей связи в стиле С++.

Нас по-прежнему интересуют три вопроса: как передать аргументы в процедуру на ассемблере, как к ним обратиться в процедуре на ассемблере и как возвратить результат?

Вначале отметим, что всегда нужно сохранять (и перед выходом из процедуры восстанавливать) содержимое регистров BP, SP, CS, DS и SS. Это делается перед вызовом процедуры. Остальные регистры нужно сохранять по необходимости, но, на

мой взгляд, хорошим тоном является сохранение и последующее восстановление всех регистров, которые подвергаются изменению внутри процедуры.

Передача аргументов в процедуру на ассемблере из программы на С осуществляется также через стек, но порядок их размещения в стеке является обратным рассмотренному ранее для связи Pascal—ассемблер. В качестве примера используем ту же задачу. После передачи управления ближнего типа процедуре на ассемблере стек должен выглядеть так, как показано на рис. 15.3, *а*.

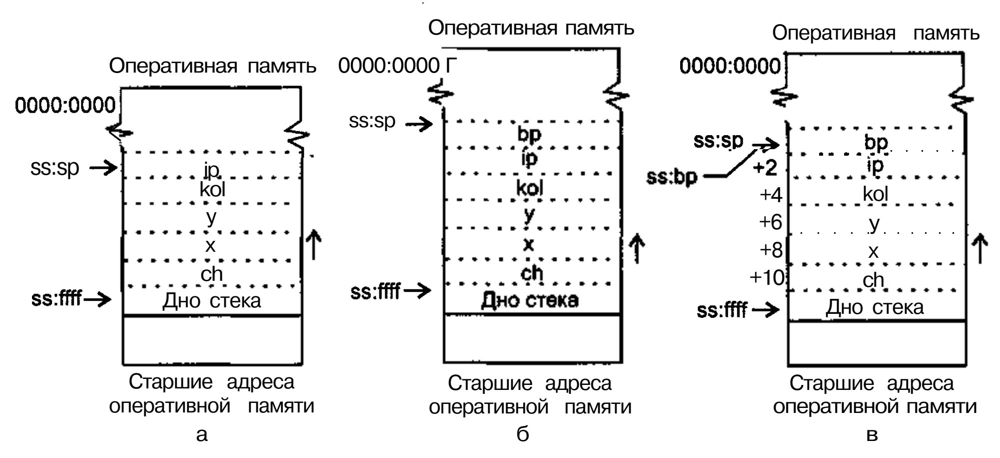


Рис. **15.3.** Изменение содержимого стека при передаче управления в связке **С—ассемблер**

Процедуры на ассемблере получают доступ к аргументам, переданным в стеке, посредством регистра BP. Принцип доступа тот же, что и рассмотренный ранее (рис. 15.3, *б*). Прежде всего в начало процедуры ассемблера необходимо вставить код пролога:

```
push    bp
mov bp,sp
```

После этого доступ к аргументам в стеке осуществляется по смещению относительно содержимого BP, например:

```
movax,[bp+4]     переписать значение ch
                 ;из стека в ax
movbx,[bp+6]     значение x - в регистр bx
```

При организации связи **С—ассемблер** можно использовать также директиву ARG. Это избавит нас от необходимости подсчитывать смещения в стеке для **доступа** к аргументам и позволит обращаться к ним просто по именам:

```
arg ch:byte;x:word;y:word;kol:word
    push bp
    mov bp,sp
...
    mov ax,[ch] ;переписать значение ch
                ;из стека в ax
    mov bx,[x]  ;значение x — в регистр bx
...
```

Чтобы не повторяться, рассмотрим, как изменятся вызываемый и вызывающий модули (листинги 15.19 и 15.20) для связи **С—ассемблер** по сравнению с листингами 15.17 и 15.18.

Листинг 15.19. Вызывающий модуль на C++

```
//prg15_19.cpp
#include <stdio.h>
#include <conio.h>
extern "C" void asmproc(char ch, unsigned x,
unsigned y,unsigned kol);
void main (void)
{
clrscr();
asmproc('a', 2, 3, 5);
asmproc('s', 9, 2, 7);
}
```

Листинг 15.20. Вызываемая процедура на ассемблере

```
;prg15_20.asm
MASM
MODEL small,c   ;модель памяти и тип кода
STACK   256
PUBLIC _asmproc ;символ подчеркивания обязателен
.code
main:
_asmproc    proc    C near c:BYTE,x:BYTE,y:BYTE.kol:WORD
    mov dh,y    ;y-координата символа в dh
    mov dl,x    ;x-координата символа в dl
    mov ah,02h  ;номер службы BIOS
    int 10h     ;вызов прерывания BIOS
    mov ah,09h  ;номер службы BIOS
    mov cx,kol  ;kol - количество "выводов" в cx
    mov bl,07h  ;маска вывода в Ы
    xor bh,bh
    mov al,c    ;c - символ в al
    int 10h     ;вызов прерывания BIOS
    ret         ;возврат из процедуры
_asmproc    endp
end main
```

Что касается передачи аргументов в связке С—ассемблер, то здесь, как видите, все довольно прозрачно. В листинге 15.20 мы используем директиву MODEL с операндом С и директиву PROC с указанием языка С. Этим мы доверяем компилятору самому сформировать коды пролога и эпилога, а также организовать обращение к переменным в стеке по их именам. Но при использовании конкретных программных средств организация такой связи выглядит намного проблематичней. Не в последнюю очередь это связано с тем, что компиляторы языков С/С++ разрабатываются множеством фирм — в отличие от Pascal, компилятор для которого выпускает практически одна фирма Borland. Это обстоятельство, на мой взгляд, — основная причина сложности связи С—ассемблер, так как каждая фирма реализует ее по-своему (хотя суть и остается практически неизменной). Поэтому, как мне кажется, нет смысла рассматривать множество частных случаев, тем более что это не является целью данной книги. Обращайтесь к документации на ваш компилятор С/С++. Во избежание излишних сложностей, по возможности применяйте ассемблерные вставки в программах на С/С++.

Как правило, компиляторы позволяют связывать модули на С/С++ и ассемблере с использованием средств командной строки. Так как этот процесс довольно хорошо стандартизован, есть смысл его рассмотреть. В качестве примера выберем компилятор С++ 5.0 фирмы Inprise (Borland). Типовая последовательность шагов выглядит примерно так.

1. Составить текст программы на С++ (см. листинг 15.19). В этой программе объявитьпроцедуруasmprocвнешней:

```
extern void asmproc(char ch, unsigned x,
    unsigned y, unsigned kol);
```

2. Выполнить трансляцию модуля С++ и получить объектный модуль:

```
bcc -c prg15_19.cpp
```

Параметр *-с* здесь означает, что выполняется только компиляция исходного файла, загрузочный модуль не создается. Результатом этого шага будет создание объектного модуля prg15_19.obj.

3. Составить текст процедуры на ассемблере (см. листинг 15.20), в которой объявить процедуру asmproc общедоступной с помощью директивы PUBLIC. Заметьте, что идентификатору asmproc предшествует символ подчеркивания (_asmproc). Компилятор С/С++ добавляет знак подчеркивания ко всем глобальным идентификаторам. Более того, некоторые компиляторы (VC++) могут кроме символа подчеркивания добавлять в конце исходного идентификатора комбинацию символов @nn, где nn означает количество байтов, занимаемых аргументами процедуры в стеке (см., например, листинг 16.2 в главе 16).

4. Выполнить трансляцию программы на ассемблере:

```
tasm prg15_20,,,
```

5. Выполнить объединение объектных модулей:

```
bcc -ms prg15_19.obj prg15_20.obj
```

Исполняемому модулю будет присвоено имя prgl5_19.exe. Параметр -ms определяет модель памяти.

Компилятор Borland С++ предоставляет другую возможность для получения загрузочного модуля. Скопируйте файл tasm.exe в каталог ..\bin пакета Borland С++. Запустите исходные файлы на трансляцию командной строкой вида

```
bcc prg15_19.cpp prg15_20.asm
```

В результате будет получен файл prgl5_19.exe. Компилятор Borland С++ всю работу организует сам: обрабатывает файл prg15_19.cpp; вызывает транслятор tasm.exe, который выполняет трансляцию файла prg15_20.asm; передает компоновщику объектные модули prg15_19.obj и prg15_20.obj. В результате создается загрузочный модуль prgl5_19.exe.

Как возвратить результат в программу на С из процедуры на ассемблере? Для этого существуют стандартные соглашения (табл. 15.3). Перед возвратом управления в программу на С в программе на ассемблере необходимо поместить результат или сформировать указатель в заданных регистрах. Для иллюстрации работы с функцией С, текст которой написан на ассемблере, рассмотрим листинги 15.21 и 15.22. В них функция, написанная на ассемблере, подсчитывает сумму элементов массива. В функцию передаются адрес массива и его длина. Результат суммирования элементов массива возвращается обратно в вызывающую программу на С.

Таблица 15.3. Возврат аргументов из процедуры на ассемблере в программу на С/С++

Тип возвращаемого значения (С++)	Место записи результата
Unsigned char	AX
Char	AX

Тип возвращаемого значения (С++)	Место записи результата
Enum	AX
Unsigned short	AX
Short	AX
Unsigned int	AX
Int	AX
Unsigned long	DX:AX
Long	DX:AX
Указатель near	AX
Указатель far	DX:AX

Листинг 15.21. Вызывающий модуль на С/С++

```
/*prg15_21.c*/
#include <stdio.h>
extern int sum_asm(int massiv[],int count);
main()
{
int mas[5]={1,2,3,4,5};
int len=5;
int sum;
sum=sum_asm(mas,en) ;
printf("%d\n",sum);
return(0);
}
```

Листинг 15.22. Вызываемая процедура на ассемблере

```
;prg15_22.asm
MASM
MODELsmall
.stack 100h
.code
public _sum_asm
_sum_asm proc C near adr_mas:word,len_mas:word
    mov ax,0
    mov cx,len_mas  ;длину массива - в cx
    mov si,adr_mas  ;адрес массива - в si
cycl:   add ax,[si] ;сложение аккумулятора с элементом массива
    add si,2        ;адресовать следующий элемент массива
    loop cycl
    ret             ;возврат из функции, результат - в ax
_sum_asm    endp
end
```

Обратите внимание на то, что листинг 15.19 содержит текст исходного файла с расширением .срр, а листинг 15.21 — с расширением .с. Соответственно, сами исходные тексты в части организации межмодульного взаимодействия также различаются.

Дополнительную информацию о связи С—ассемблер вы найдете в [18].

Несколько слов об организации связи С—ассемблер для компилятора Visual С/С++. Один из подходов к ее реализации — оформление ассемблерной программы в виде функции из библиотеки DLL. В этом случае можно уйти от «фирменных» проблем связи, возникающих из-за несовпадения форматов информации

в объектных файлах Borland и Microsoft, и писать функции TASM и MASM для связи с программами Visual C/C++. Разработка библиотек DLL для ассемблерных программ описана в [8].

Таким образом, мы рассмотрели связь модулей на языках высокого уровня с модулями на ассемблере. Это обсуждение не могло быть полным из-за потенциальной глубины самой проблемы. Материал данного раздела можно рассматривать лишь как введение (хотя и достаточно подробное) в проблему межъязыковых отношений. Главная цель этого первого шага — разобраться с принципами. Следующим шагом в направлении реализации связи с ассемблером будет изучение документации на конкретный язык высокого уровня для выяснения особенностей настройки и требований конкретной среды программирования.

Итоги

8 Язык ассемблера содержит довольно мощные средства поддержки модульного подхода в рамках структурного программирования. В языке ассемблера эта технология поддерживается в основном с помощью механизма процедур и, частично, механизма макроподстановок.

Л Гибкость интерфейса между процедурами достигается за счет разнообразия вариантов передачи аргументов в процедуру и возвращения результатов. Для этого могут использоваться регистры, общие области памяти, стек, директивы EXTRN и PUBLIC.

и Компилятор MASM имеет мощное средство для работы с процедурами в виде директив PROTO и INVOKE. Их использование существенно облегчает вызов процедур и передачу в них параметров. Сочетание возможностей этих директив с возможностями директивы PROC позволяет организовывать корректный и более надежный вызов процедур, перекладывая заботу о формировании кодов пролога и эпилога на транслятор.

9. Средства TASM поддерживают связи между языками. Ключевой момент при этом — организация обмена данными. Обмен данными между процедурами на языках высокого уровня и ассемблера производится через стек. Для доступа к аргументам используются регистр BP или (что более удобно) директива ARG.

▒ Можно доверить компилятору самому формировать коды пролога и эпилога, указав язык в директиве MODEL. Кроме того, указание языка позволяет задействовать символические имена аргументов, переданных процедуре в стеке, вместо прямого использования регистра BP для доступа к ним. Тем самым повышаются мобильность разрабатываемых программ и устойчивость их к ошибкам.

* Для возвращения результата в программу на языке высокого уровня необходимо использовать конкретные регистры. Через них можно передать как сами данные, так и указатели.

и Довольно эффективным для организации связи С—ассемблер может быть подход, при котором ассемблерный код оформляется в виде функций библиотек DLL.

Глава 16

Создание Windows-приложений на ассемблере

- Особенности разработки Windows-приложений
- Каркасное Windows-приложение на языке C/C++
- Каркасное Windows-приложение на ассемблере
- Средства TASM для разработки Windows-приложений
- Расширенное программирование на ассемблере для Win32 API
- Ресурсы Windows-приложений на языке ассемблера
- Меню в Windows-приложениях
- Перерисовка изображения
- Использование окон диалога
- Работа с графикой

Программирование для операционной системы Windows всегда было занятием не из легких, и на эту тему написано немало книг. Сказать что-то новое довольно трудно, но и обойти эту тему в книге, посвященной одному из языков программирования, на современном этапе развития вычислительной техники было бы не совсем правильно. Сегодня трудно найти компьютер, на котором бы не была установлена одна из версий Windows, если, конечно, на нем не стоит что-то из «мира» UNIX. Но в этом учебнике речь будет идти исключительно о программировании для Windows на платформе Intel.

В подавляющем большинстве книг о программировании для Windows изложение, как правило, ведется на базе языков C/C++, реже — на базе Pascal. А что же

ассемблер — в стороне? Конечно, нет! Мы не раз обращали ваше внимание на правильное понимание места ассемблера в архитектуре компьютера. Любая программа на языке самого высокого уровня по сути представляет собой последовательность машинных кодов. А раз так, то всегда остается теоретическая возможность написать ту же программу, но уже на языке ассемблера. Чем можно обосновать необходимость разработки Windows-приложений на языке ассемблера? Приведем следующие аргументы:

- язык ассемблера позволяет программисту полностью контролировать создаваемый им программный код и оптимизировать его по своему усмотрению;
- компиляторы языков высокого уровня помещают в загрузочный модуль программы избыточную информацию, поэтому эквивалентные исполняемые модули, исходный текст которых написан на ассемблере, имеют в несколько раз меньший размер;
- при программировании на ассемблере сохраняется полный доступ к аппаратным ресурсам компьютера;
- приложение, написанное на ассемблере, как правило, быстрее загружается в оперативную память компьютера;
- приложение, написанное на ассемблере, обладает, как правило, более высокой скоростью работы и ответа на действия пользователя.

Разумеется, эти аргументы не следует воспринимать, как некоторую рекламную кампанию в поддержку языка ассемблера. Тем более что компиляторы языков высокого уровня постоянно совершенствуются и подчас способны создавать код, весьма близкий по эффективности к ассемблерному. По этой причине приведенные аргументы не являются бесспорными. И все же нельзя забывать о том, что существует бесконечное множество прикладных задач, ждущих своей очереди на компьютерную реализацию. Далеко не все из этих задач требуют тяжеловесных средств разработки — многие из них могут быть изящно исполнены на языке ассемблера, не теряя привлекательности, например, оконных Windows-приложений.

Перед началом обсуждения поясним, в чем состоит разница между программированием для DOS и для Windows. Операционные системы MS-DOS и Windows поддерживают две совершенно разные идеологии программирования. В чем разница? Программа DOS после своего запуска должна быть постоянно активной. Если ей, к примеру, требуется получить очередную порцию данных с устройства ввода-вывода, то она сама должна выполнять соответствующие запросы к операционной системе. При этом программа DOS работает по определенному алгоритму, она всегда знает, что и когда ей следует делать. В Windows все наоборот. Программа пассивна. После запуска она ждет, когда ей уделит внимание операционная система. Операционная система делает это посылкой специально оформленных групп данных, называемых *сообщениями*. Сообщения могут быть разного типа, они функционируют в системе довольно хаотично, и приложение не знает, какого типа сообщение придет следующим. Отсюда следует, что логика построения Windows-приложения должна обеспечивать корректную и предсказуемую работу при поступлении сообщений любого типа. Тут можно провести определенную аналогию между механизмом сообщений Windows и механизмом прерываний в архи-

тектуре IBM PC. Для нормального функционирования своей программы программист должен уметь эффективно использовать функции интерфейса прикладного программирования (Application Program Interface, API) операционной системы.

Windows поддерживает два типа приложений.

- *Оконное приложение* строится на базе специального набора функций API, составляющих графический интерфейс пользователя (Graphic User Interface, GUI). Оконное приложение представляет собой программу, которая весь вывод на экран производит в графическом виде. Первым результатом работы оконного приложения является отображение на экране специального объекта — окна. После того как окно появилось на экране, вся работа приложения направлена на то, чтобы поддерживать его в актуальном состоянии.
- *Неоконное приложение,* также называемое *консольным*, представляет собой программу, работающую в текстовом режиме. Работа консольного приложения напоминает работу программы MS-DOS. Но это лишь внешнее впечатление. Поддержка работы консольного приложения обеспечивается специальными функциями Windows.

Вся разница между двумя типами Windows-приложений состоит в том, с каким типом информации они работают. Основной тип приложений в Windows — оконные, поэтому с них мы и начнем знакомство с процессом разработки программ для этой операционной системы.

Программирование оконных Windows-приложений

Любое оконное Windows-приложение имеет типовую структуру, основу которой составляет так называемое *каркасное приложение,* содержащее минимально необходимый для функционирования полноценного Windows-приложения программный код. Не случайно во всех источниках в качестве первого Windows-приложения рекомендуется изучать и исследовать работу некоторого каркасного приложения, так как именно оно отражает основные особенности взаимодействия программы с операционной системой Windows. Более того, написанное и однажды отлаженное каркасное Windows-приложение используется и в дальнейшем в качестве основы для написания любого другого значительно более сложного приложения.

Изложение материала будем иллюстрировать программами на двух языках — C/C++ и ассемблере. Такой подход значительно облегчает понимание технологии написания Windows-приложений на ассемблере. На его основе можно даже выработать некую методику, которая позволит конвертировать многие полезные программы на C/C++ в функционально эквивалентные программы на ассемблере.

Перед началом изложения отметим некоторые его характерные черты.

- Теоретический и практический материал главы будет отражать особенности разработки программ для 32-разрядных операционных систем Windows, к которым относятся Windows 95/98 и Windows NT/2000/XP. Хотя архитектуры этих систем в большей или меньшей степени различаются, их объединяет 32-разрядный программный интерфейс — Win32 API. Он представляет собой набор

функций, к которым может обращаться приложение. Основная идея Win32 API — обеспечение переносимости программ между различными программно-аппаратными платформами.

- Несмотря на то что изложение будет вестись довольно подробно, мы не сможем описать все детали процесса построения Windows-приложения. Но в этом нет ничего страшного, так как в настоящее время доступно довольно много источников, где это сделано с необходимой степенью детализации. Неподготовленному читателю можно посоветовать подобрать другой источник, где начальный уровень программирования для Windows изложен с соответствующей степенью детализации. При этом ему совсем не нужно влезать в дебри. Вполне достаточно достичь уровня понимания логики работы каркасного Windows-приложения, и можно снова браться за данный учебник.
- Для изучения материала этой главы и его практического использования в дальнейшей работе мало иметь только один пакет TASM. Кроме него также необходимы пакеты инструментальных средств разработки приложений на языке C/C++, например от Microsoft или Borland. В том и другом пакетах имеются все необходимые средства для разработки Windows-приложений. Пакет TASM, в отличие от этих пакетов, не обладает такими средствами, поэтому программисту приходится заимствовать их в том или ином виде в пакетах C/C++.

Каркасное Windows-приложение на C/C++

Обсуждение вопросов программирования для Windows на ассемблере начнем с обсуждения программы на языке C/C++. Не нужно удивляться такому подходу — «цель оправдывает средства». Нам необходимо, во-первых, понять общие принципы построения оконных Windows-приложений. Во-вторых, разобраться с тем, какие средства ассемблера при этом используются. Добиваться этих целей без предварительного обсуждения нецелесообразно. Сделаем это мягко и ненавязчиво, через рассказ о построении минимального Windows-приложения на языке C/C++. В ходе его разработки мы введем необходимую терминологию и сможем больше внимания уделить логике работы Windows-приложения, а не деталям его реализации. После этого мы с относительной легкостью разработаем эквивалентное приложение на ассемблере.

Приступая к разработке первого (и не только) Windows-приложения, важно понимать, что сам язык программирования мало влияет на его общую структуру. Это обстоятельство, кстати, и позволит нам чуть позже с относительной легкостью сменить инструментальное средство разработки Windows-приложений с C/C++ на ассемблер.

Минимальное приложение Windows состоит из трех частей:

- главной функции;
- цикла обработки сообщений;
- оконной функции.

Выполнение любого оконного Windows-приложения начинается с *главной функции*. Она содержит код, осуществляющий настройку (инициализацию) приложе-

ния в среде Windows. Видимым для пользователя результатом работы главной функции является появление на экране графического объекта в виде окна. Последним действием кода главной функции является создание *цикла обработки сообщений*. После его создания приложение становится пассивным и начинает взаимодействовать с внешним миром посредством специальным образом оформленных данных — *сообщений*. Обработка поступающих приложению сообщений осуществляется специальной функцией, называемой *оконной*. Оконная функция уникальна тем, что может быть вызвана только из операционной системы, а не из приложения, которое ее содержит (функция обратного вызова). Тело оконной функции имеет определенную структуру, о которой мы поговорим далее. Таким образом, Windows-приложение, как минимум, должно состоять из трех перечисленных элементов. В листинге 16.1 приведен вариант минимального приложения на языке С/С++.

Листинг 16.1. Каркасное Windows-приложение на языке С/С++

```
#include<windows.h>
LRESULT CALLBACK WindowProc(HWND , UINT , WPARAM , LPARAM);
char szClassWindow[] = "Каркасное приложение";  /*Имя класса окна*/
int WINAPI WinMain(HINSTANCE hlnst, HINSTANCE hPrevInst,
  LPSTR lpszCmdLine, int nCmdShow)
{
    HWND hWnd;
    MSG lpMsg;
    WNDCLASSEX wcl;
/* Определение класса окна */
wcl.cbSize = sizeof (wcl);          //длина структуры WNDCLASSEXA
wcl.style = CS_HREDRAW|CS_VREDRAW;  //CS (Class Style) - стиль класса окна
wcl.lpfnWndProc = WindowProc;       //адрес функции окна
wcl.cbClsExtra =0;                 //для внутреннего использования Windows
wcl.cbWndExtra = 0;                //для внутреннего использования Windows
wcl.hInstance = hlnst;              //дескриптор данного приложения
wcl.hIcon = LoadIconA(NULL, IDI_APPLICATION);  //стандартная иконка
wcl.hCursor = LoadCursorA(NULL, IDC_ARROW);    //стандартный курсор
wcl.hbrBackground =(HBRUSH)GetStockObject (WHITE_BRUSH);//определить
                                                        //заполнение
wcl.lpszMenuName = NULL;             //окна белым цветом без меню
wcl.lpszClassName = szClassWindow;  //имя класса окна
wcl.hIconSm=NULL ;           //дескриптор значка, связываемого с классом окна
//зарегистрировать класс окна
if (!RegisterClassEx (&wcl))
return 0;
//создать окно и присвоить дескриптор окна переменной hWnd
hWnd=CreateWindowEx(
    0,  //расширенный стиль окна
    szClassWindow,                      //имя класса окна
    "Каркас программы для Win32 на C++",//заголовок окна
    WS_OVERLAPPEDWINDOW,                //стиль окна
    CW_USEDEFAULT,                 //X-координата верхнего левого угла окна
    CW_USEDEFAULT,                 //Y-координата верхнего левого угла окна
    CW_USEDEFAULT,                 //ширина окна
    CW_USEDEFAULT,                 //высота окна
    NULL,                          //дескриптор родительского окна
    NULL,                          //дескриптор меню окна
    hlnst,                       //идентификатор приложения, создавшего окно
    NULL);                         //указатель на область данных приложения
//показать окно и перерисовать содержимое
ShowWindow (hWnd, nCmdShow);
UpdateWindow (hWnd);
```

продолжение ⇨

Листинг 16.1 *(продолжение)*

```
/* запустить цикл обработки сообщений */
while (GetMessage(&lpMsg, NULL, 0, 0))
{
    TranslateMessage(&lpMsg);    //разрешить использование клавиатуры
    DispatchMessage(&lpMsg);    //вернуть управление Windows
}
    return lpMsg.wParam;
}                                    //конец WinMain
LRESULT CALLBACK WindowProc (HWND hWnd, UINT message,
  WPARAM wParam, LPARAM IParam)
//Функция WndProc вызывается операционной системой Windows
//и получает в качестве параметров сообщения из очереди
//сообщений данного приложения
{
    swi tch(message)
    {
    case WM_DESTROY: /* завершение программы */
    PostQuitMessage (0);
    break;
    default:
    //Сюда попадают все сообщения,
    //не обрабатываемые в данной оконной функции.
    //Далее эти сообщения направляются обратно Windows
    //на обработку по умолчанию
    return DefWindowProc (hWnd, message, wParam, lParam);
}
return 0;
}
```

Рассмотрим более подробно суть действий, выполняемых каждым из трех элементов Windows-приложения. В листинге 16.1 видно, что минимальное Windows-приложение на языке C++ состоит из двух функций: главной — WinMain и оконной — WindowProc. Цель WinMain — сообщить системе о новом для нее приложении, его свойствах и особенностях. Для достижения этой цели WinMain выполняет определенную последовательность действий.

1. Определяет и регистрирует класс окна приложения.
2. Создает и отображает окно приложения зарегистрированного класса.
3. Создает и запускает цикл обработки сообщений для приложения.
4. Завершает программу при получении оконной функцией соответствующего сообщения.

Оконная функция получает все сообщения, предназначенные данному окну, и обрабатывает их, возможно, вызывая для этого другие функции.

Видимая часть работы каркасного приложения заключается в создании нового окна на экране. Оно отвечает всем требованиям стандартного окна Windows-приложения, то есть его можно развернуть, свернуть, изменить размер, переместить в другое место экрана и т. д. Для этого, как вы увидите, нам не придется написать ни строчки кода, а нужно будет всего лишь удовлетворить определенные требования, предъявляемые к приложениям со стороны Windows.

Мы не будем больше обсуждать код листинга 16.1, так как это довольно подробно и полно сделано в других источниках. Взамен этого мы пойдем по другому пути — заглянем за «фасад» приведенного Windows-приложения и посмотрим на работу, выполняемую компилятором по формированию соответствующего испол-

няемого кода. Причем сделать это целесообразно на двух этапах: в процессе формирования объектного кода и после формирования загрузочного модуля.

В процессе формирования объектного модуля компилятор преобразует исходный текст на языке C/C++ в эквивалентный текст на языке ассемблера. В контексте нашего обсуждения это достаточно ценная информация. Для того чтобы получить такой текст, необходимо в командной строке компилятора задать специальный ключ ДА (см. make-файл в каталоге данной главы с программой prg16_1.cpp среди файлов, прилагаемых к книге[1]):

```
cl /FA ... prg16_1.cpp
```

Полученный текст на ассемблере ценен тем, что в нем каждой строке исходного текста программы на C/C++ сопоставляется текст на ассемблере. В листинге 16.2 приведен фрагмент этого файла (prg16_1.asm), а полный его текст находится в каталоге \Lesson16\prg16_1\c среди файлов, прилагаемых к книге.

Листинг 16.2. Фрагмент ассемблерного представления исходного файла prg16_1.cpp

```
TITLE   prg16_1.cpp
.386P
include listing.inc
if @Version gt 510
.model FLAT
else
_TEXT   SEGMENT PARA USE32 PUBLIC 'CODE'
...
endif
PUBLIC  ?szClassWindow@@3PADA   ;szClassWindow
_DATA   SEGMENT
...
_DATA   ENDS
PUBLIC  _WinMain@16
PUBLIC  ?WindowProc@@YGJPAUHWND_@@IIJ@Z ;WindowProc
EXTRN   _imp_GetMessageA@16:NEAR
...
_TEXT   SEGMENT
; File prg19_1.cpp
...
_WinMain@16 PROC NEAR
;Строка 5
    push    ebp
    mov ebp, esp
    sub esp, 80 ;00000050H
    push    ebx
    push    esi
    push    edi
; Строка 10
...
; Строка 12
    mov DWORD PTR_wcl$[ebp+8], OFFSET FLAT:?WindowProc@@YGJPAUHWND_@@IIJ@Z
;WindowProc
; Строка 13
    mov DWORD PTR_wcl$[ebp+12], 0
...
; Строка 16
    push    32512   ; 00007f00H
```

продолжение ⇨

[1] Все прилагаемые к книге файлы можно найти по адресу http://www.piter.com/download. — *Примеч. ред.*

Листинг 16.2 *(продолжение)*

```
    push    0
    call    DWORD PTR_imp_LoadIconA@8
    mov     DWORD PTR_wcl$[ebp+24], eax
...
; Строка 18
    push    0
    call    DWORD PTR_imp_GetStockObject@4
    mov     DWORD PTR_wcl$[ebp+32], eax
...
; Строка 24
    lea     eax, DWORD PTR_wcl$[ebp]
    push    eax
    call    DWORD PTR_imp_RegisterClassExA@4
    movzx   eax,ax
    test    eax, eax
    jne     $L29705
; Строка 25
    xor     eax, eax
    jmp     $L29694
; Строка 27
$L29705:
; Строка 39
    push    0
    mov     eax, DWORD PTR_hInst$[ebp]
    push    eax
    push    0
    push    0
    push    -2147483648  ;80000000H
    push    -2147483648  ;80000000H
    push    -2147483648  ;80000000H
    push    -2147483648  ;80000000H
    push    13565952;00cf0000H
    push    OFFSET FLAT:$SG29710
    push    OFFSET FLAT:?szClassWindow@@3PADA   ;szClassWindow
    push    0
    call    DWORD PTR_imp_CreateWindowExA@48
    mov     DWORD PTR_hWnd$[ebp], eax
; Строка 41
    mov     eax, DWORD PTR_nCmdShow$[ebp]
    pusheax
    mov     eax, DWORD PTR_hWnd$[ebp]
    push    eax
    call    DWORD PTR_imp_ShowWindow@8
; Строка 42
    mov     eax, DWORD PTR_hWnd$[ebp]
    push    eax
    call    DWORD PTR_imp_UpdateWindow@4
; Строка 44
$L29712:
    push    0
    push    0
    push    0
    lea     eax, DWORD PTR_lpMsg$[ebp]
    push    eax
    call    DWORD PTR_imp_GetMessageA@16
    test    eax, eax
    je      $L29713
; Строка 46
    lea     eax, DWORD PTR_lpMsg$[ebp]
    push    eax
    call    DWORD PTR_imp_TranslateMessage@4
; Строка 47
```

```
    lea     eax, DWORD PTR_lpMsg$[ebp]
    push    eax
    call    DWORD PTR_imp_DispatchMessageA@4
; Строка 48
    jmp     $L29712
$L29713:
; Строка 49
    raov    eax, DWORD PTR_lpMsg$[ebp+8]
    jmp     $L29694
; Строка 50
$L29694:
    pop     edi
    pop     esi
    pop     ebx
    leave
    ret     16        ;00000010H
_WinMain@16 ENDP
...
?WindowProc@@YGJPAUHWND_@@IIJ@Z PROC NEAR    ;WindowProc
; Строка 55
    push    ebp
    mov     ebp, esp
    sub     esp, 4
    push    ebx
    push    esi
    push    edi
; Строка 56
    mov     eax, DWORD PTR_message$[ebp]
    mov     DWORD PTR -4+[ebp], eax
    jmp     $L29719
; Строка 58
$L29723:
; Строка 59
    push    0
    call    DWORD PTR_imp_PostQuitMessage@4
; Строка 60
    jmp     $L29720
; Строка 61
$L29724:
; Строка 64
    mov     eax, DWORD PTR_lParam$[ebp]
    push    eax
    mov     eax, DWORD PTR_wParam$[ebp]
    push    eax
    mov     eax, DWORD PTR_message$[ebp]
    push    eax
    mov     eax, DWORD PTR_hWnd$[ebp]
    push    eax
    call    DWORD PTR_imp_DefWindowProcA@16
    jmp     $L29718
; Строка 65
    jmp     $L29720
$L29719:
    cmp     DWORD PTR -4+[ebp], 2
    je      $L29723
    jmp     $L29724
$L29720:
; Строка 66
    xor     eax, eax
    jmp     $L29718
; Строка 67
$L29718:
    pop     edi
    pop     esi
```

продолжение ⇨

Листинг 16.2 *(продолжение)*

```
    pop ebx
    leave
    ret 16    ;00000010H
?WindowProc@@YGJPAUHWND_@@IIJ@Z ENDP;WindowProc
_TEXT    ENDS
END
```

Беглый взгляд на код листинга 16.2 показывает, что он оформлен как полноценная программа на ассемблере. Код программы был сгенерирован компилятором Visual C++ 6.0 от Microsoft, поэтому транслятор MASM может сделать из него загрузочный модуль без дополнительного редактирования. Для этого компания Microsoft даже подготовила специальный включаемый файл listing.inc (он находится в каталоге пакета VC++ 4.0:..\Msdev\include, а также среди файлов, прилагаемых к книге).

Таким образом, имея исходный файл Windows-приложения на языке C/C++, можно получить текст на языке ассемблера. Теоретически, на его основе впоследствии можно сформировать функционально эквивалентный исполняемый модуль. Более того, если задаться такой целью, то не составит большого труда переделать его для обработки транслятором TASM 5.0.

Но не все так просто. Готов расстроить читателя, но мы изложили очень упрощенный подход к разработке Windows-приложения на ассемблере. Полученный код можно рассматривать лишь как схему (шаблон) такого приложения. Реально все несколько сложнее. Прежде чем мы объясним имеющиеся проблемы, проведем еще один эксперимент — дизассемблируем исполняемый модуль программы prg16_1.cpp. Причем сделать это нужно тем дизассемблером, который «понимает» интерфейс Win32 API. В данной книге для этой цели был использован дизассемблер IDA. Дизассемблированный с его помощью файл можно сохранить как листинг (.lst) и как исходный текст ассемблера (.asm). Это говорит о том, что IDA также пытается по загрузочному модулю построить файл, пригодный для повторной трансляции. В начале дизассемблированного им текста даже приводятся рекомендации по использованию соответствующих параметров транслятора TASM.

В контексте нашего изложения наибольший интерес представляет файл листинга (.lst), формируемый IDA. Анализ его содержимого позволяет изнутри посмотреть на результаты процесса формирования исполняемого модуля транслятором и редактором связей. Файл листинга в своей левой части содержит колонку с адресными смещениями команд. Все метки и символические имена в дизассемблированном тексте формируются с учетом этих смещений, поэтому данный файл удобно использовать для анализа.

Файл листинга IDA имеет большой размер, поэтому привести весь его текст в книге невозможно, да это и не нужно. Для нас интерес представляют лишь отдельные фрагменты этого файла (листинг 16.3). Полностью Дизассемблированный код программы prg16_1.cpp находится среди фалов, прилагаемых к книге.

Листинг 16.3. Фрагменты дизассемблированного кода каркасного Windows-приложения (дизассемблер IDA)

```
; ===================================================================
; Этот файл сгенерирован интерактивным дизассемблером(IDA)
; Copyright (c) 1997 by DataRescue sprl, <ida@datarescue.com>
; ===================================================================
```

```
...
00401000 ;этот файл должен компилироваться следующей командной строкой: TASM
/ml/m5
00401000    p386
00401000    model flat
...
00401000_WinMain@16 proc near    ;CODE   XREF: start+146_p
...
00401000    push    ebp
00401001    mov     ebp,esp
00401003    sub     esp,50h
00401006    push    ebx
00401007    push    esi
00401008    push    edi
...
00401037    push    0
00401039    call    ds:LoadIconA
0040103F    mov     [ebp+var_38] , eax
00401042    push    7F00h
00401047    push    0
00401049    call    ds:LoadCursorA
...
00401072    lea     eax, [ebp+var_50]
00401075    push    eax
00401076    call    ds:RegisterClassExA
0040107C    movzx   eax, ax
...
0040108E loc_40108E:       ;CODE   XREF: WinMain@16+81_j
0040108E    push    0
00401090    mov     eax, [ebp+arg_0]
00401093    push    eax
00401094    push    0
00401096    push    0
00401098    push    80000000h
0040109D    push    80000000h
004010A2    push    80000000h
004010A7    push    80000000h
004010AC    push    0CF0000h
004010B1    push    offset  unk_404034
004010B6    push    offset  unk_404020
004010BB    push    0
004010BD    call    ds:CreateWindowExA
...
004010CE    call    ds:ShowWindow
00401004    mov     eax, [ebp+var_4]
004010D7    push    eax
004010D8    call    ds:UpdateWindow
004010DE
004010DE loc_4010DE:       ; CODE  XREF: WinMain@16+10A_j
004010DE    push    0
004010E0    push    0
004010E2    push    0
004010E4    lea     eax, [ebp+var_20]
004010E7    push    eax
004010E8    call    ds:GetMessageA
004010EE    test    eax,eax
004010F0    jz      loc_40110F
004010F6    lea     eax, [ebp+var_20]
004010F9    push    eax
004010FA    call    ds:TranslateMessage
00401100    lea     eax, [ebp+var_20]
00401103    push    eax
00401104    call    ds:DispatchMessageA
0040110A    jmp     loc_4010DE
```

продолжение ⇒

Листинг 16.3 *(продолжение)*

```
...
00401117 loc_401117:     ;CODE   XREF:_WinMain@16+89_j
00401117   pop    edi
00401118   pop    esi
00401119   pop    ebx
0040111A   leave
0040111B   retn    10h
0040111B_WinMain@16 endp
...
;оконная процедура
...
00401132 loc_401132:     ;CODE  XREF: .text:00401163_j
00401132   push    0
00401134   call    ds:PostQuitMessage
0040113A   jmp     loc_40116E
...
0040114E   push    eax
0040114F   call    ds:DefWindowProcA
...
00401175   pop     edi
00401176   pop     esi
00401177   pop     ebx
00401178   leave
00401179   retn    10h
...
00401180
00401180 public start
00401180 start  proc near
...
004011A6    call  ds:GetVersion   ;получить текущую версию Windows
004011A6   ;и информацию о текущей операционной платформе
...
00401205    call     ds:GetCommandLineA
0040120B    mov     ds:dword_4050B0, eax
00401210    call     _crtGetEnvironmentStringsA
...
004012A0    call     ds:GetStartupInfoA
...
004012BF    call     ds:GetModuleHandleA
004012C5    push    eax
004012C6    call    _WinMain@16
004012CB    push    eax
004012CC    call    _exit
...
0040130E    pop    esi
0040130F    pop    ebx
00401310    mov    esp, ebp
00401312    pop    ebp
00401313   retn
00401313 start  endp;sp = -88h
00401313
...
00401380_exit proc near ;CODE   XREF: start+A9_p
00401380   ; start+14C_p
00401380
...
00401389    call    sub_4013C0   ;doexit
...
00401391    _exit   endp
...
004013C0 ;    п о д п р о г р а м м а doexit
004013C0 ; Атрибуты: Статическая библиотечная функция
```

```
004013C0 sub_4013C0 proc near   ; CODE  XREF:_exit+9_p
...
004013D1    call    ds:GetCurrentProcess
004013D7    push    eax
00401308    call    ds:TerminateProcess
...
00401458    push    esi
00401459    call    ds:ExitProcess
...
00401462     retn
00401462  sub_4013C0  endp
...
0040610C ; Импорт из  KERNEL32.dll
0040610C
0040610C     extrn GetModuleFileNameA:dword   ;  DATA XREF:_setargv+12_r
0040610C    ;_NMSG_WRITE+77_r
...
00406188 Импорт из USER32.dll
00406188     extrn PostQuitMessage:dword ;DATA XREF:  .text:00401134_r
0040618C     extrn DispatchMessageA:dword;DATA XREF:_WinMain@16+104_r
...
004061B4   end start
```

Текст листинга 16.3 дает богатую пищу для размышлений. Его обсуждение нужно начать с последней строки. По правилам ассемблера, в последней директиве программы END должна стоять метка первой исполняемой команды. В нашем случае это метка start. Если теперь посмотреть на место в тексте, куда она ссылается, то мы увидим, что в данной строке содержится директива PROC, обозначающая начало процедуры с именем start. Можно сделать вывод, что исполнение программы начинается именно с этого места. Обратите внимание на то, что в теле процедуры start содержатся многочисленные вызовы функций.

В листинге 16.3 показаны только вызовы функций API, но если вы посмотрите полный вариант листинга 16.3, находящийся среди файлов к книге, то увидите, что в нем содержатся вызовы множества других функций. Названия этих функций начинаются символом подчеркивания (_), в отличие от названий функций API, которым предшествует префикс переопределения сегмента, например: ds:GetVersion. Задержимся немного и посмотрим внимательно на листинг 16.1, в частности, нас интересуют функции, к которым идет обращение. Видно, что в исходном тексте Windows-приложения, написанном на C/C++, нет и намека на какие-либо функции, за исключением функций API. Более того, вызов многих из функций API, которые присутствуют в дизассемблированном тексте, отсутствует в листинге 16.1. Отсюда следует естественный вывод о том, что их вставил компилятор C/C++ и, очевидно, они предназначены для инициализации среды, в которой будет функционировать программа. А так ли они необходимы? Забегая вперед, скажем, что нет. Но и отказаться от них нельзя, так как они фактически «навязываются» нам компилятором C/C++. В этом и состоит одна из главных причин того, что исполняемые модули на ассемблере по размеру в несколько раз меньше функционально эквивалентных модулей на языках высокого уровня. Можно сказать, что приложение, написанное на ассемблере, не содержит избыточного кода. Отметьте еще один характерный момент относительно функций API. Кодовый сегмент исполняемого модуля содержит только команды дальнего перехода к этим функциям. Сами же функции находятся в отдельном файле — библиотеке DLL. В конце

листинга 16.3 содержится таблица, в которой описано местонахождение всех импортируемых функций API.

Проследим за действиями, которые производит код исполняемого модуля, формируемый компилятором Visual C++ для инициализации Windows-приложения. Для этого откройте файл prg16_1.lst с дизассемблированным текстом программы на C/C++, находящийся среди прилагаемых к книге файлов, и переместитесь на начало процедуры start.

1. Вызов функции API GetVersion[1] для определения текущей версии Windows и некоторой другой информации о системе. В настоящее время существует более информативная версия данной функции — GetVersionEx.
2. Инициализация кучи. Используется как для динамического выделения памяти функциями языков C/C++, так и для организации ввода-вывода на низком уровне.
3. Вызов функции API GetCommandLineA для получения указателя на командную строку, с помощью которой было вызвано данное приложение.
4. Вызов функции API GetEnvironmentStringsA для получения указателя на блок с переменными окружения.
5. Инициализация глобальных переменных периода выполнения.
6. Вызов функции API GetStartupInfoA для заполнения структуры, поля которой определяют свойства основного окна приложения.
7. Вызов функции API GetModuleHandleA для получения базового адреса, по которому загружен данный исполняемый модуль.
8. Вызов функции WinMain. Как можно судить по способу вызова и внутреннему названию, _WinMain@16 не является функцией API. Это локальная функция данного исполняемого модуля. Территориально она расположена в начале дизассемблированного текста (см. листинг 16.3). Позже обсудим ее подробнее.

Главный вывод приведенных рассуждений — текст процедуры start исполняемого модуля не соответствует исходному тексту программы на C/C++. В него добавлены локальные функции и функции Win32 API, которые вызываются в исполняемом модуле, например GetVersion, GetCommandLineA и т. д. Это и есть так называемый *стартовый код* программы. Если продолжить изучение этого стартового кода в файле prg16_1.lst, то можно увидеть вызов функции _WinMain@16. Найдем теперь тело этой функции в тексте. Даже беглое сравнение функции _WinMain@16 (см. листинг 16.3) и функции WinMain (см. листинг 16.1) показывает, что мы нашли место, где содержится код ассемблера, функционально эквивалентный коду на C/C++. В частности, хорошо видно, что в _WinMain@16 вызываются именно те функции (и никакие другие), что и в функции WinMain программы на C/C++.

Мы не будем сейчас обсуждать порядок и цель вызова каждой из этих функций, как это делалось для стартовой процедуры, по той причине, что следующий

[1] Описания всех упомянутых в книге функций и структур данных Win32 API приводятся лишь частично. Полную информацию о них вы можете получить в многочисленных источниках, в которых рассматриваются вопросы разработки Windows-приложений. Предпочтительно использовать первоисточник, находящийся в Интернете по адресу: http://www.microsoft.com/msdn/.

наш шаг — это создание программы на языке ассемблера. В процессе ее разработки мы и рассмотрим эти вопросы достаточно подробно. Сейчас нам важно сделать вывод, что функция WinMain не имеет прямого отношения к функциям Win32 API. Эта функция используется лишь для того, чтобы компилятор мог сгенерировать код, выполняющий инициализацию приложения.

Кстати, если вы внимательно посмотрите на листинг 16.3, то без труда обнаружите другие фрагменты кода исполняемого файла, прямо соответствующие тексту исходного файла на С/С++. Например, в качестве упражнения найдите текст оконной функции.

В заключение обсуждения обратите внимание на код завершения программы:

```
004012C6        call _WinMain@16
004012CB        push eax
004012CC        call _exit
```

Видно, что для завершения приложения вызывается процедура _exit. Код, который она содержит, является обязательным для корректного завершения любого Windows-приложения. Более подробно мы его обсудим при разработке каркасного Windows-приложения на языке ассемблера.

Каркасное Windows-приложение на ассемблере

Одним из главных критериев выбора языка разработки Windows-приложения является наличие в нем средств, способных поддержать строго определенную последовательность шагов. Язык ассемблера является универсальным языком и пригоден для реализации любых задач, поэтому можно смело предположить, что на нем можно написать также любое Windows-приложение. Материал, изложенный ранее, наглядно это доказал. Более того, стали видны некоторые подробности кода, который должно содержать Windows-приложение на ассемблере. Но мало написать сам текст Windows-приложения, необходимо знать средства пакета транслятора, специально предназначенные для разработки таких приложений, и уметь пользоваться этими средствами.

В листинге 16.4 приведен текст каркасного приложения на ассемблере, функционально эквивалентного Windows-приложению на С/С++ (см. листинг 16.1). Если не обращать внимания на особенности оформления кода, обусловленные требованиями синтаксиса ассемблера, то хорошо видно, что на уровне функций его структура аналогична рассмотренному ранее Windows-приложению на С/С++.

Листинг 16.4. Каркасное Windows-приложение на ассемблере

```
;Пример каркасного приложения для Win32
.386
locals ;разрешает применение локальных меток в программе
.model flat, STDCALL   ;модель памяти flat
;STDCALL - передача параметров в стиле С (справа налево),
;  вызываемая процедура чистит за собой стек
include windowA.inc;включаемый файл с описаниями базовых структур
                   ;и константт Win32
;Объявление внешними используемых в данной программе
;функций Win32 (ASCII):
    extrn   GetModuleHandleA:PROC
```

продолжение

Листинг 16.4 *(продолжение)*

```
<10>   extrn   GetVersionExA:PROC
<11>   extrn   GetCommandLineA:PROC
<12>   extrn   GetEnvironmentStringsA:PROC
<13>   extrn   GetEnvironmentStringsA:PROC
<14>   extrn   GetStartupInfoA:PROC
<15>   extrn   LoadIconA:PROC
<16>   extrn   LoadCursorA:PROC
<17>   extrn   GetStockObject:PROC
<18>   extrn   RegisterClassExA:PROC
<19>   extrn   CreateWindowExA:PROC
<20>   extrn   ShowWindow:PROC
<21>   extrn   UpdateWindow:PROC
<22>   extrn   GetMessageA:PROC
<23>   extrn   TranslateMessage:PROC
<24>   extrn   DispatchMessageA:PROC
<25>   extrn   ExitProcess:PROC
<26>   extrn   PostQuitMessage:PROC
<27>   extrn   DefWindowProcA:PROC
<28>   extrn   PlaySoundA:PROC
<29>   extrn   ReleaseDC:PROC
<30>   extrn   TextOutA:PROC
<31>   extrn   GetDC:PROC
<32>   extrn   BeginPaint:PROC
<33>   extrn   EndPaint:PROC
<34>   ;объявление оконной функции объектом,
       ;видимым за пределами данного кода
<35>   public  WindowProc
<36>   .data
<37>   hwnd    dd  0
<38>   hInst   dd  0
<39>   hdc     dd  0
<40>   ;lpVersionInformation   OSVERSIONINFO   <?>
<41>   wcl WNDCLASSEX  <?>
<42>   message MSG <?>
<43>   ps  PAINTSTRUCT <?>
<44>   szClassName db  'Приложение Win32', 0
<45>   szTitleName db  'Каркасное приложение Win32 на ассемблере', 0
<46>   MesWindow   db  'Привет! Как звук в наушниках.
                   Не нравится? - Отключите!!!'
<47>   MesWindowLen=   $-MesWindow
<48>   playFileCreate  db  'create.wav', 0
<49>   playFilePaint   db  'paint.wav', 0
<50>   playFileDestroy db  'destroy.wav', 0
<51>   .code
<52>   start   proc near
<53>   ;точка входа в программу:
<54>   ;начало стартового кода
<55>   ;вызовы расположенных ниже функций
       ;можно при необходимости раскомментировать,
<56>   ;но они не являются обязательными в данной программе
<57>   ;вызов BOOL GetVersionEx(LPOSVERSIONINFO IpVersionlnformation)
<58>   ;   push    offset IpVersionlnformation
<59>   ;   call    GetVersionExA
<60>   ;далее можно вставить код
       ;для анализа информации о версии Windows
<61>   ;вызов LPTSTR GetCommandLine(VOID) - получить указатель
       ;на командную строку
<62>   ;   call    GetCommandLineA ;в регистре eax адрес
<63>   ;вызов LPVOID GetEnvironmentStrings (VOID) - получить указатель
       ;на блок с переменными окружения
<64>   ;   call    GetEnvironmentStringsA  ;в регистре eax адрес
<65>   ;вызов VOID GetStartupInfo(LPSTARTUPINFO lpStartupInfo)  ;указатель
       ;на структуру STARTUPINFO
```

```
;   push     offset lpStartupInfo
;   call     GetStartupInfoA
;вызов HMODULE GetModuleHandleA (LPCTSTR lpModuleName)
    push     NULL;0->GetModuleHandle
    call     GetModuleHandleA     ;получить значение базового адреса,
    mov hInst, eax                ;по которому загружен модуль.
;далее hlnst будет использоваться в качестве
;дескриптора данного приложения
;конец стартового кода
WinMain:
;определить класс окна
;ATOM RegisterClassEx(CONST WNDCLASSEX *lpWndClassEx),
;где *lpWndClassEx - адрес структуры WndClassEx
;для начала инициализируем поля структуры WndClassEx
    mov     wcl.cbSize, type WNDCLASSEX ;размер структуры
                                        ;в wcl.cbSize
    mov     wcl.style, CS_HREDRAW+CS_VREDRAW
    mov     wcl.lpfnWndProc, offset WindowProc   ;адрес оконной
                                                 ;процедуры
    mov     wcl.cbClsExtra, 0
    mov     wcl.cbWndExtra, 0
    mov     eax, hlnst
;дескриптор приложения в поле hInstance структуры wcl
    mov     wcl.hInstance, eax
;готовим вызов
;HICON LoadIcon (HINSTANCE hlnstance, LPCTSTR lpIconName)
    push     IDI_APPLICATION ;стандартный значок
    push     0    ;NULL
    call     LoadIconA
    mov      wcl.hIcon, eax   ;дескриптор значка в поле hIcon
                              ;структуры wcl
;готовим вызов
;HCURSOR LoadCursorA (HINSTANCE hlnstance, LPCTSTR lpCursorName)
    push     IDC_ARROW    ;стандартный курсор - стрелка
    push     0
    call     LoadCursorA
    mov wcl.hCursor, eax;дескриптор курсора в поле hCursor
                        структуры wcl
;определим цвет фона окна - белый
;готовим вызов HGDIOBJ GetStockObject(int fnObject)
    push     WHITE_BRUSH
    call     GetStockObject
    mov     wcl.hbrBackground, eax
    mov     dword   ptr wcl.lpszMenuName, 0 ;без главного меню
    mov     dword ptr wcl.lpszClassName, offset szClassName ;имя
                                                            ;класса окна
raov    wcl.hIconSm, 0
;регистрируем класс окна - готовим вызов
;RegisterClassExA  (&wndclass)
    push     offset wcl
    call     RegisterClassExA
    test    ax, ax  ;проверить на успех регистрации класса окна
    jz      end_cycl_msg;неудача
создаем окно:
;готовим вызов
;HWND CreateWindowExA(DWORD dwExStyle,  LPCTSTR lpClassName,
;LPCTSTR lpWindowName, DWORD dwStyle, int x, int y, int nWidth,
;int nHeight, HWND hWndParent, HMENU hMenu, HANDLE hlnstance,
;LPVOID lpParam)
    push     0    ;lpParam
    push     hlnst    ;hInstance
    push     NULL;menu
    push     NULL;parent hwnd
    push     CW_USEDEFAULT    ;высота окна
    push     CW_USEDEFAULT    ;ширина окна
```

продолжение

Листинг 16.4 *(продолжение)*

```
    push    CW_USEDEFAULT   ;координата у левого верхнего угла окна
    push    CW_USEDEFAULT   ;координата х левого верхнего угла
    push    WS_OVERLAPPEDWINDOW ;стиль окна
    push    offset szTitleName  ;строка заголовка окна
    push    offset szClassName  ;имя класса окна
    push    NULL
    call    CreateWindowExA
    mov     hwnd, eax           ;hwnd - дескриптор окна
;показать окно:
;готовим вызов BOOL ShowWindow( HWND hWnd, int nCmdShow )
    push    SW_SHOWNORMAL
    push    hwnd
    call    ShowWindow
;перерисовываем содержимое окна
;готовим вызов BOOL UpdateWindow( HWND hWnd )
    push    hwnd
    call    UpdateWindow
;запускаем цикл сообщений:
;готовим вызов BOOL GetMessageA( LPMSG lpMsg, HWND hWnd,
;UINT wMsgFilterMin, UINT wMsgFilterMax )
cycl_msg:
    push    0
    push    0
    push    NULL
    push    offset message
    call    GetMessageA
    cmp     ax, 0
    je      end_cycl_msg
;трансляция ввода с клавиатуры
;готовим вызов BOOL TranslateMessage( CONST MSG *lpMsg )
    push    offset message
    call    TranslateMessage
;отправим сообщение оконной процедуре
;готовим вызов LONG DispatchMessage( CONST MSG *lpmsg)
    push    offset message
    call    DispatchMessageA
    jmp     cycl_msg
end_cycl_msg:

;выход из приложения
;готовим вызов VOID ExitProcess( UINT uExitCode )
    push    NULL
    call    ExitProcess
start   endp
;..................WindowProc---------------------------------
WindowProc  proc
arg @@hwnd:DWORD, @@mes:DWORD, @@wparam:DWORD, @@lparam:DWORD
uses ebx, edi, esi  ;эти регистры обязательно должны сохраняться
local   @@hdc:DWORD
    cmp @@mes, WM_DESTROY
    je  wmdestroy
    cmp @@mes, WM_CREATE
    je  wmcreate
    cmp @@mes, WM_PAINT
    je  wmpaint
    jmp default
wmcreate:
;обозначим создание окна звуковым эффектом
;готовим вызов функции
;BOOL PlaySound(LPCSTR pszSound, HMODULE hmod, DWORD fdwSound)
    push    SND_SYNC+SND_FILENAME
    push    NULL
    push    offset playFileCreate
```

```
        call    PlaySoundA
        mov     eax, 0  ;возвращаемое значение - 0
        jmp     exit_wndproc
wmpaint:
        push    SND_SYNC+SND_FILENAME
        push    NULL
        push    offset playFilePaint
        call    PlaySoundA
;получим контекст устройства
;HDC BeginPaint( HWND hwnd, LPPAINTSTRUCT lpPaint )
        push    offset ps
        push    @@hwnd
        call    BeginPaint
        mov     @@hdc, eax
;выведем строку текста в окно
;BOOL TextOut( HDC hdc, int nXStart. int nYStart,
;LPCTSTR lpString, int cbString )
        push    MesWindowLen
        push    offset MesWindow
        push    100 .
        push    10
        push    @@hdc
        call    TextOutA
;освободить контекст
;BOOL EndPaint( HWND hWnd,  CONST PAINTSTRUCT *lpPaint )
        push    offset ps
        push    @@hdc
        call    EndPaint
        mov     eax, 0  ;возвращаемое значение - 0
        jmp     exit_wndproc
wmdestroy:
        push    SND_SYNC+SND_FILENAME
        push    NULL
        push    offset playFileDestroy
        call    PlaySoundA
;послать сообщение WM_QUIT
;готовим вызов VOID PostQuitMessage( int nExitCode )
        push    0
        call    PostQuitMessage
        mov     eax, 0  ;возвращаемое значение - 0
        jmp     exit_wndproc
default:
;обработка по умолчанию
;готовим вызов LRESULT DefWindowProc( HWND hWnd, UINT Msg,
;WPARAM wParam, LPARAM lParam )
        push    @@lparam
        push    @@wparam
        push    @@mes
        push    @@hwnd
        call    DefWindowProcA
        jmp     exit_wndproc
;...
exit_wndproc:
        ret
WindowProc  endp
end start
```

Каркасное Windows-приложение на ассемблере содержит один сегмент данных **.data** и один сегмент кода **.code**. Сегмент стека в исходных текстах Windows-приложений непосредственно описывать не нужно. Windows выделяет для стека объем памяти, размер которого задан программистом в файле с расширением **.def**. Текст листинга 16.4 довольно большой. Поэтому для обсуждения разобьем его коммен-

тариями на характерные фрагменты, каждый из которых затем поясним с необходимой степенью детализации.

Строка 3. Все символические имена в программе на ассемблере по умолчанию являются глобальными. Задание директивы LOCALS включает в трансляторе механизм контроля областей видимости имен и позволяет использовать в программе *локальные* имена (см. главу 10).

Строка 4. Директива .MODEL задает модель сегментации (flat) и стиль генерации (см. главу 15) кода при входе в процедуры программы и выходе из них (stdcall). Модель памяти flat обозначает плоскую модель памяти. В соответствии с этой моделью компилятор создает программу, которая содержит один 32-разрядный сегмент для данных и кода программы. Код загрузочного модуля, генерируемый с параметром flat, будет работать на процессорах i386 и выше. По этой причине директиве .MODEL должна предшествовать одна из директив: .386, .486 или .586. Указание этой модели памяти заставляет компоновщик создать исполняемый файл с расширением .exe. В программе с плоской моделью памяти используется адресация программного кода и данных типа near. Это же касается и атрибута расстояния в директиве PROC, который также имеет тип near. Параметр stdcall определяет порядок передачи параметров через стек так, как это принято в языке C/C++, то есть справа налево. Этот параметр задает генерацию кода эпилога, который очищает стек по завершении работы процедуры (в стиле языка Pascal) от аргументов, переданных в эту процедуру при вызове. Такая передача параметров очень удобна при разработке Windows-приложений: программист может помещать параметры в стек в прямом порядке, и ему не нужно заботиться об очистке стека после окончания работы процедуры.

Строка 7. Директива INCLUDE включает в программу файл windowA.inc. Для чего нужен этот файл? Вы, наверняка, имеете некоторый опыт разработки Windows-приложений на C/C++ и знаете, что функции Win32 API в качестве передаваемых им параметров используют множество констант и данных определенной структуры. В пакете компилятора C/C++ эти данные расположены в заголовочных файлах, совокупность которых можно рассматривать как дерево с корнем в файле windows.h. Для того чтобы эти описания стали доступны приложению на C/C++, в его начало включается директива #include <windows.h> (см. листинг 16.1). Windows-приложение на ассемблере также использует вызовы функций Win32 API, поэтому в нем должны быть выполнены аналогичные действия по определению необходимых констант и структур. Но просто взять из пакета C/C++ и затем использовать в программе на ассемблере файл windows.h и связанные с ним файлы напрямую нельзя. Причина банальна: транслятор ассемблера не понимает синтаксиса C/C++. Что делать? Пакет TASM предоставляет в распоряжение программиста утилиты h2ash.exe и h2ash32.exe, которые должны помочь ему конвертировать содержимое включаемых файлов на языке C/C++ во включаемые файлы ассемблера. Однако, исходя из опыта, не стоит питать особых надежд на эффективность их работы. Поэтому проще формировать ассемблерный включаемый файл с описанием констант и структур Windows самостоятельно. Кстати, пакет TASM 5.0 предоставляет в распоряжение программиста вариант такого включаемого файла win32.inc. Его можно найти среди прилагаемых к книге файлов в каталоге к данной главе, но

пользоваться им нужно с осторожностью, так как его содержимое не совсем актуально. Например, файл win32.inc содержит описание структуры WNDCLASS, но в нем отсутствует расширенный вариант этой структуры WNDCLASSEX, который требуется в расширенном варианте функции RegisterClassExA. Как решить проблему актуальности описаний? Проще всего сделать это, используя включаемые файлы последних на текущий момент времени версий компилятора C/C++ (сейчас это, например, VC++ 6.0 и 7.0). Это один из основных источников подобной информации, актуальность которой к тому же гарантирована разработчиком компилятора, являющегося одновременно и создателем операционной системы Windows. Поэтому программирование на ассемблере для Windows предполагает, что вы хорошо умеете ориентироваться во включаемых файлах компилятора C/C++. Полностью пытаться конвертировать эти файлы не имеет смысла. Более правильно извлекать из них по мере необходимости описание нужных данных, приводить их в соответствие требованиям синтаксиса ассемблера и после этого записывать в некоторый свой файл, который будет играть роль файла windows.h. Для Windows-приложений данной главы вам предлагается вариант такого включаемого файла — windowA.inc, который включается в текст приложения директивой include windowA.inc. Файл windowA.inc имеется среди файлов, прилагаемых к книге. Его можно взять за основу для дальнейшей работы и наращивать по мере необходимости.

Строки 9-33. Функции Win32 API, используемые в программе, должны быть объявлены внешними с помощью директивы EXTRN. Это необходимо для того, чтобы компилятор мог сгенерировать правильный код, так как тела функций Win32 API содержатся в библиотеках DLL системы Windows.

Имейте в виду, что в различных источниках встречаются разные названия, казалось бы, одной и той же функции. Необходимо правильно понимать их. Например, уже упомянутая функция RegisterClass может иметь названия RegisterClassExA или RegisterClassExW. На самом деле две последние функции являются более современными вариантами RegisterClass. Для разрешения подобных коллизий приходится обращаться к первоисточникам — документации с описанием функций Win32 API (помните, что она также может оказаться устаревшей). Самый лучший источник такой информации — включаемые файлы компилятора C/C++. Например, описание функции RegisterClass(ExA) содержится в файле winuser.h. В листинге 16.5 приведены строки из этого файла.

Листинг 16.5. Фрагмент файла winuser.h (VC++ 6.0)

```
WINUSERAPI ATOM WINAPI RegisterClassA(CONST WNDCLASSA *lpWndClass);
WINUSERAPI ATOM WINAPI RegisterClassW(CONST WNDCLASSW *lpWndClass);
#ifdef UNICODE
#define RegisterClass RegisterClassW
#else
#define RegisterClass RegisterClassA
#endif // !UNICODE

#if(WINVER >= 0x0400)
WINUSERAPI ATOM WINAPI RegisterClassExA(CONST WNDCLAS5EXA *);
   WINUSERAPI ATOM WINAPI RegisterClassExW(CONST WNDCLASSEXW *);
   #ifdef UNICODE
   #define Regi sterClassEx RegisterClassExW
```

продолжение ⇨

Листинг 16.5 *(продолжение)*

```
<13>    #else
<14>    #define RegisterClassEx RegisterClassExA
<15>    tfendif    // !UNICODE
```

Последние версии операционных систем Windows поддерживают две системы кодировки символов: однобайтную (ANSI) и двухбайтную (UNICODE). Поддержка кодировки UNICODE была введена Microsoft, чтобы облегчить локализацию программных продуктов на неанглоязычном рынке. Операционная система Windows NT поддерживает только кодировку UNICODE. Операционная система Windows 95/98 имеет довольно плохо скрытое наследие MS-DOS и не поддерживает в своей внутренней работе кодировку UNICODE (она поддерживается лишь на уровне функций Win32 API). Для работы с обеими кодировками программный интерфейс Win32 API имеет два варианта функций. Эти функции различаются последним символом в названии. Если это A, то данная функция работает в кодировке ANSI, если W, то функция работает в кодировке UNICODE. Следующий момент, требующий пояснения, — наличие суффикса Ex в названиях функций. Объяснение этому можно найти в листинге 16.5. Директива условной компиляции (строка 8) проверяет текущую версию операционной системы. Если это Windows 95/98 или NT, то можно использовать расширенные (с суффиксами Ex) варианты функций Win32 API. Они обладают дополнительными возможностями по сравнению со старыми вариантами функций Win32 API. Исчерпывающим источником информации по функциям Win32 API (и не только) является MSDN (Microsoft Developer Network — информационная система поддержки разработчика по продуктам Microsoft), его можно найти в Интернете по адресу http://www.microsoft.com/msdn/.

Строка 35. В соответствии с соглашениями операционной системы Windows, оконная функция приложения должна быть видимой за пределами приложения, в котором она описана. Это связано с тем, что оконная функция вызывается самой операционной системой Windows при поступлении сообщений для данного приложения. По этой причине оконная функция нашего каркасного приложения объявлена общей (public).

Строки 36-50 содержат описание сегмента данных, в котором определяются переменные и экземпляры структур, используемые в каркасном приложении.

Упомянем еще об одном важном техническом моменте программирования для операционной системы Windows — соглашениях об именовании различных программных объектов. Обозначениям в Windows придается большое значение. Это объясняется сложностью разрабатываемых систем, а также тем, что пользователи и разработчики должны понимать друг друга при создании как программных продуктов, так и документации к ним. Без согласованных правил по обозначению тех или иных объектов им не обойтись. Единых требований на этот счет нет. В настоящее время де-факто в качестве системы обозначений принята так называемая форма *венгерской записи*. Более подробные сведения о ней с указанием первоисточника приведены в файле \Lesson16\Hungarian Notation.htm, который находится среди файлов, прилагаемых к книге. Не стоит принимать венгерскую форму записи за некую догму. Это всего лишь одна из возможных систем обозначений. При желании вы можете ввести свою собственную систему. Для того чтобы заставить

компилятор ассемблера различать строчные и прописные буквы, необходимо указать ключ /ml в командной строке.

Перед рассмотрением сегмента кода обратите внимание на его начало (строка 51). В нем отсутствуют привычные команды загрузки сегментного регистра данных. Загрузчик Windows самостоятельно загружает сегментные регистры, при этом учитывается требуемая модель памяти (директива .model). В первой части главы мы упоминали, что для запуска приложения под управлением Windows необходимо выполнить нескольких шагов, в которых выполняется вызов ряда функций Win32 API. Перечислим их.

1. Выполнение стартового кода.
2. Выполнение главной функции (в C/C++ — вызов функции WinMain), которая выполняет следующие действия:
   1) регистрирует класс окна;
   2) создает окно;
   3) отображает окно;
   4) запускает цикл обработки сообщений;
   5) завершает выполнение приложения.
3. Организация обработки сообщений в оконной процедуре.

А как вызывать функции Win32 API в программе на ассемблере? Вызов этих функций осуществляется аналогично вызову внешних функций (см. главу 15). Передача всех параметров осуществляется через стек. По этой причине важно знать размеры передаваемых величин. Здесь проявляется полезность венгерской системы обозначений. О размере переменной можно судить по ее названию. К тому же в Win32 большинство переменных имеет размер двойного слова (четыре байта). В соответствии с параметром stdcall директивы .MODEL параметры в стек должны помещаться справа налево, то есть первым в стек идет последний параметр функции.

Продолжим рассмотрение каркасного Windows-приложения (см. листинг 16.4).

Стартовый код

Структура нашего каркасного Windows-приложения строится в соответствии с аналогичным приложением на C/C++ (см. листинг 16.1) и его дизассемблированным вариантом (см. листинг 16.3). Разрабатывая приложение на C/C++, мы «прячемся за спину» компилятора, доверяя ему часть работы. Программируя на ассемблере, мы лишаемся этой «широкой спины» и вынуждены всю работу делать сами. Что представляет собой эта работа, видно из листинга 16.3. Первый шаг Windows-приложения заключается в исполнении *стартового* кода (строки 54-73). Стартовый код представляет собой последовательный вызов функций Win32 API. Вы можете экспериментировать с ними при разработке своих программ (например, для доступа к информации о параметрах командной строки или переменных окружения), но в общем случае использовать большинство из них не обязательно. Чтобы продемонстрировать это, в листинге 16.4 вызов некоторых функций стартового кода закомментирован. В этой программе из всего стартового кода мы оставили лишь вызов функции GetModuleHandleA (который, кстати, тоже можно за-

комментировать без потери работоспособности программы). Она предназначена для идентификации исполняемого файла в адресном пространстве процесса. Здесь нужно кое-что пояснить.

В литературе по платформе Win32 часто встречаются такие понятия, как *процесс* и *поток. Процесс* (приложение) представляет собой экземпляр программы, загруженной в память для выполнения. Процесс инертен, он просто владеет пространством памяти в 4 Гбайт. В этом пространстве содержатся код и данные, другие ресурсы, загружаемые в адресное пространство процесса. В качестве ресурсов могут быть, в свою очередь, исполняемые файлы или библиотеки DLL. Для того чтобы процесс исполнялся, в нем должен быть создан *поток,* который, собственно, и отвечает за исполнение кода, содержащегося в адресном пространстве процесса.

Зачем такие сложности? Дело в том, что Win32 по сравнению со своими предшественниками кроме процессной многозадачности поддерживает еще и потоковую многозадачность, при которой в рамках одного процесса параллельно запускаются несколько потоков. Таким образом, в рамках одной программы виртуально выполняются несколько фрагментов кода. В принципе, имея несколько процессоров, возможно реальное распараллеливание вычислительного процесса в рамках одного приложения.

Из-за того что в адресное пространство процесса можно загрузить несколько файлов, для работы с ними требуется некий механизм их однозначной идентификации. При загрузке исполняемого файла (или библиотеки DLL) в адресное пространство процесса ему присваивается уникальный номер. Этот номер передается операционной системой в качестве первого параметра hInst (Handle Instance — описатель экземпляра) функции WinMain в программе на языке С/С++. Это значение используется впоследствии при вызове многих функций Win32 API, загружающих те или иные ресурсы для данной программы. Что касается значения, присваиваемого hlnst, то оно равно базовому адресу в адресном пространстве процесса, по которому загружен данный файл с расширением .exe. Выяснить его значение можно с помощью функции GetModuleHandle. При вызове этой функции в качестве ее единственного параметра передается адрес ASCIIZ-строки с именем исполняемого файла (библиотеки DLL), базовый адрес загрузки (значение hlnst) которого мы хотим получить. Если вызвать функцию GetModuleHandle, передав ей значение NULL, то мы получим значение hlnst для текущей программы, что, кстати, и делает стартовый код (см. листинг 16.3) в программе листинга 16.1. Важно отметить, что эта идентификация актуальна только в рамках одного процесса.

Как правило, в системе существует по крайней мере несколько процессов, владеющих своим четырехгигабайтным адресным пространством и имеющих исполняемые файлы с одинаковыми идентификаторами. Причина в том, что в Win32 адресные пространства процессов разделены, и каждый из этих процессов полагает, что он в системе один.

Фрагмент кода с вызовом GetModuleHandleA выглядит так:

```
.data
hinst   dd  0
...
.code
    push    0
    call    GetModuleHandleA
    mov     hinst, eax
```

После этого кода вызывается главная функция приложения — WinMain (см. листинг 16.3).

Главная функция

Основная задача главной функции (строки 74-162) оконного Windows-приложения состоит в правильной инициализации программы и корректном ее завершении. Правильная инициализация приложения предполагает выполнение ряда предопределенных шагов. Обсудим их подробнее.

Регистрация класса окна

Регистрация класса окна выполняется в строках 75-108. Под *классом окна* понимается совокупность присущих ему характеристик, таких как стиль его границ, формы указателя мыши, значков, цвет фона, наличие меню, адрес оконной процедуры, обрабатывающей сообщения этого окна. Класс окна можно впоследствии использовать для создания окон приложения функцией CreateWindow. Характеристики окна описываются с помощью специальной структуры WNDCLASS (или ее расширенного варианта WNDCLASSEX в Win32).

В сегменте данных в строке 41 листинга 16.4 определен экземпляр структуры WNDCLASSEX — wcl. Первое поле структуры WNDCLASSEX cbSize должно содержать длину структуры. Команда в строке 80 загружает в это поле соответствующее значение. В поле style можно определять стиль границ окна и его поведение при перерисовке. Значение стиля является целочисленным и формируется из констант. Каждая константа означает некоторую предопределенную характеристику. Включаемый файл winuser.h компилятора VC++ содержит символические названия этих констант. Эти же константы, но уже в соответствии с требованиями синтаксиса ассемблера, описаны в файле windowsA.inc. В строке 81 листинга 16.4 комбинация констант CS_HREDRAW и CS_VREDRAW определяет необходимость полной перерисовки окна при изменении его вертикального или горизонтального размера.

В строке 80 в поле lpfnWndProc записывается адрес оконной функции. С помощью этой функции все окна, созданные позднее функцией CreateWindow на основе класса, для которого выполняется регистрация, будут обрабатывать посланные им сообщения.

Поля cbClsExtra и cbWndExtra служат для указания количества байтов, дополнительно резервируемых в структуре класса окна WNDPROC и структуре параметров окна, которая поддерживается внутри самой системы Windows. Обычно эти поля инициализированы нулевыми значениями.

Строка 85 формирует в ноле hInstance дескриптор приложения, полученный ранее функцией GetModuleHandleA.

В строках 86-95 в поля hIcon и hCursor загружаются дескрипторы значка и указателя мыши. После запуска приложения значок будет отображаться на панели задач Windows и в левом верхнем углу окна приложения, а указатель мыши появится в области окна. Значки и указатели мыши представляют собой ресурсы и находятся в отдельных файлах. Windows предоставляет в распоряжение программиста ряд стандартных изображений указателей мыши и значков. В файле winuser.h содержатся символические имена констант, обозначающих стандартные указатели мыши и значки. Обратите внимание на первый параметр функций LoadCursorA

и LoadIconA — hInst. Это дескриптор приложения, содержащий базовый адрес ресурса значка или указателя мыши, загруженного в процесс. Если используются их стандартные изображения, то параметр hlnst равен NULL

Команды в строках 96-99 формируют поле hbrBackground, которое должно содержать значение дескриптора кисти. *Кисть* представляет собой ресурс в виде шаблона пикселов, которым закрашивается некоторый объект, в данном случае — фон окна приложения некоторого класса. Для получения такого дескриптора необходимо использовать функцию GetStockObject. В качестве параметра ей передается имя нужной кисти. В файле wingdi.h содержатся символические имена констант, определяющих стандартные кисти.

В строке 101 в поле lpszMenuName записывается указатель на ASCIIZ-строку с именем меню. Если меню не используется (как в нашем случае), то в поле записывается значение NULL.

Поле hIconSm в строке 103 можно рассматривать как альтернативу полю hIcon. В него помещается дескриптор значка, который будет связан с данным классом окна. Если поле hlconSm нулевое, то система будет использовать значок, определенный полем hlcon.

И последнее действие при описании класса окна — присвоение данному классу уникального имени. Это имя описано в виде ASCIIZ-строки в поле szClassName сегмента данных, и указатель на него формируется в поле lpszClassName (строка 102).

После инициализации структуры необходимо зарегистрировать класс окна в системе. Это действие выполняется в строках 105-108 с помощью функции RegisterClassExA, которой в качестве параметра передается указатель на структуру WNDCLASSEX.

Необходимо заметить, что после того как класс окна зарегистрирован, структура WNDCLASSEX больше не нужна. У вас появляется возможность сэкономить немного памяти. Это можно сделать, используя предоставляемый ассемблером тип данных объединения либо инициализируя поля структуры в стеке с последующим их выталкиванием. Здесь есть широкое поле для экспериментов. Дерзайте!

Создание окна

После того как класс окна описан и зарегистрирован в системе, приложение на его основе может создать множество различных окон. Создание окна (строки 109-126) выполняется функцией Win32 API CreateWindowEx. Для этого ей нужно передать множество параметров. Назначение их мы рассмотрим далее. В качестве результата функция возвращает уникальный дескриптор окна hWnd, который необходимо сохранить (строки 37, 125, 126).

У читателя, видимо, возникает вопрос, почему для создания окна необходимы два шага: сначала определение класса окна, а лишь затем непосредственно его создание. Даже если не рассматривать этот вопрос в контексте концепции объектно-ориентированного программирования, двухэтапный процесс создания окна очень удобен для практической работы. В качестве наглядного примера приведем кнопки редактора MS Word. На самом деле это маленькие окна, созданные на базе одного класса. Они используют одну оконную функцию, которая обрабатывает сообщения, посланные этим окнам. Какой именно кнопке послано сообщение, оконная

функция определяет по полю hwnd в структуре сообщения (см. далее). Наличие одного класса для всех кнопок гарантирует их одинаковое поведение. В то же время, каждая из кнопок может отличаться внешним видом. Но это уже дополнительные свойства конкретного окна-кнопки, которые задаются при создании экземпляра окна параметрами функции CreateWindowEx.

Рассмотрим на примере Windows-приложения (см. листинг 16.4) задание значений параметров функции CreateWindowEx (строки 113–124).

Строка 113. Параметр lpParam используется при создании окна для передачи данных или указателя на них в оконную функцию. Делается это следующим образом. Все параметры, передаваемые функции CreateWindowEx, сохраняются в создаваемой Windows внутренней структуре CREATESTRUCT. Поля этой структуры идентичны параметрам функции CreateWindowEx. Указатель на структуру CREATESTRUCT передается оконной функции при обработке сообщения WM_CREATE. Сам указатель находится в поле lParam сообщения. Значение параметра IpParam функции CreateWindowEx находится в поле lpCreateParams структуры CREATESTRUCT.

Строка 114. Параметр hInst — дескриптор приложения, создающего окно.

Строка 115. Параметр hMenu — дескриптор главного меню окна. Так как в данном варианте Windows-приложения меню у окна отсутствует, то параметр hMenu имеет нулевое значение.

Строка 116. Параметр hWndParent — дескриптор родительского окна. Между двумя окнами Windows-приложения можно устанавливать родовые отношения. Дочернее окно всегда должно появляться в области родительского окна. Так как у нас подобных отношений нет, то значение передаваемого параметра нулевое.

Строки 117-120. Параметры, загружаемые в стек этими строками, задают начальные координаты и размеры окна приложения на экране. Константа CW_USEDEFAULT (80000000h), определенная в файле winuser.h, позволяет «попросить» Windows использовать значения этих параметров по умолчанию.

Строка 121. Параметр dwStyle определяет стиль окна приложения. Значение этого параметра задается константой или комбинацией констант. Символические имена этих констант описаны во включаемом файле winuser.h (Visual C++ 6.0). В нашем приложении используется значение WS_OVERLAPPEDWINDOW (00000000h), которое задает стиль обычного перекрывающегося окна с рамкой, имеющего системное меню, кнопки свертывания, развертывания и закрытия окна в правом верхнем углу.

Строки 122 и 123. Параметры szTitleName и szClassName — их значения являются указателями на ASCIIZ-строки (строки 44-45) с именем класса окна и текстом, помещаемым в заголовок окна.

Строка 124. Параметр dwExStyle позволяет задать дополнительные стили окна. В Win32 API существует две реализации функции создания окна: стандартная CreateWindowA и расширенная CreateWindowExA. Для их вызова используются одинаковые параметры, за исключением последнего — dwExStyle. При использовании функции CreateWindowA параметр в стек помещать не требуется, при использовании CreateWindowExA параметр dwExStyle должен быть помещен в стек последним. Дополнительные стили можно задействовать вместе со стилями, задаваемыми параметром dwStyle.

Строка 125. Вызов функции CreateWindowExA. В качестве результата функция возвращает дескриптор окна hWnd. Он имеет уникальное значение и является одним из важнейших описателей объектов приложения. Он передается как параметр многим функциям Win32 API и как значение полей в некоторых структурах. Одновременно в приложении может быть создано и совместно существовать несколько окон, поэтому с помощью дескриптора hWnd Windows однозначно идентифицирует то окно, для работы с которым вызывается та или иная функция Win32 API. Другой важный результат работы функции CreateWindowExA — посылка асинхронного сообщения WM_CREATE в оконную функцию приложения. В нашей программе обработка этого сообщения заключается в вызове функции PlaySoundA, которая воспроизводит звуковой файл. Более подробно о сообщениях и их обработке поговорим далее.

Отображение окна

В случае успешного выполнения функции CreateWindowExA требуемое окно будет создано, но пока это произойдет лишь внутри самой системы Windows — на экране это новое окно пока еще не отобразится. Для того чтобы созданное окно появилось на экране, необходимо применить еще одну функцию Win32 API — ShowWindowA. В качестве параметров этой функции передаются дескриптор hWnd окна, которое необходимо отобразить на экране, и константа, задающая начальный вид окна на экране. В зависимости от значения последнего параметра окно отображается в стандартном виде, развернутым на весь экран или свернутым в значок. Описание символических констант, задающих начальный вид окна на экране, содержится во включаемом файле winuser.h. Строки 127-131 нашего Windows-приложения задают отображение окна в стандартном виде.

Если окно создано, то в него нужно что-то выводить — текст или графику. Здесь есть несколько тонких моментов, выяснение которых позволит нам «подняться сразу на несколько ступеней» в понимании принципов работы Windows. Поэтому задержимся немного. Тем более что в литературе этим моментам не всегда уделяется достаточно внимания — в лучшем случае десяток строк, большая часть которых уходит на обсуждение функций и их параметров, а не на разъяснение сути происходящих при этом процессов. Ход наших рассуждений будем подкреплять практическими наблюдениями за работой приложения. Для этого можно использовать утилиты, которые позволят нам наблюдать за сообщениями, циркулирующими в системе, и вызовами функций API. В качестве таких программных средств можно использовать утилиты, входящие в комплект компиляторов для языка C/C++. Например, в состав Visual C++ входит утилита Spy++ (шпион), которая позволяет наблюдать за приложениями, находящимися в данный момент в системе. Неплохие результаты для Windows-приложений дает дизассемблер W32Dasm, информацию о котором можно найти по адресу http://www.expage.com/page/w32dasm.

Если у вас нет ничего подобного под рукой, отчаиваться не стоит. Можно провести необходимые исследования опытным путем, просто меняя логику работы приложения. Суть этой методики состоит в комбинировании последовательности выполнения функций Win32 API, воспроизводящих звуковые файлы и выводя-

щих текст на экран. При этом необходимо перемещать те или иные фрагменты программы друг относительно друга. Но это еще не все. Само воспроизведение относительно длительных звуковых файлов выполняется в определенном режиме. Обратите внимание на значение параметра fdwSound, который мы передаем функции PlaySoundA. В нашем случае это комбинация двух констант, одна из которых (SND_SYNC) означает, что функция не должна возвращать управление до тех пор, пока не закончится воспроизведение звукового файла. Если комбинировать вывод текстового сообщения и воспроизведение звукового файла, то это дает некоторую задержку, в ходе которой глаз может воспринять момент появления текстового сообщения на экране.

Используем изложенную методику для изучения проблемы вывода информации (не обязательно текста) на экран. В нашей программе мы будем экспериментировать с выводом некоторого сообщения на экран, взяв за основу строки 189–205. Выполним с ними несколько манипуляций и обсудим их результаты.

Вырежем строки 189-205 из оконной процедуры WindowProc и вставим их в промежутке между функциями CreateWindowExA и ShowWindowA. Перетранслируем заново исходный текст приложения, получим новый вариант исполняемого модуля и запустим его. Поведение нашей программы будет следующим: воспроизводится звуковой файл create.wav (что означает обработку асинхронного сообщения WM_CREATE оконной функцией), появляется пустое окно (отработала функция ShowWindowA), воспроизводится звуковой файл paint.wav (это означает, что в оконную функцию было передано сообщение WM_PAINT). На экране вы ничего не увидите, кроме пустого окна. Почему окно оказалось пустым несмотря на то, что мы в него поместили текстовое сообщение функцией TextOut? О причинах сложившейся ситуации можно привести некоторые соображения.

- Работа функции ShowWindowA заключается лишь в отображении созданного функцией CreateWindowExA окна заданного класса и последующем заполнении фона этого окна цветом кисти, указанной в соответствующем поле структуры WNDCLASS.

- Вывод в область окна становится возможным лишь после отображения окна на экране.

- В системе Windows циркулируют два типа сообщений: синхронные и асинхронные. Их различие — в приоритетах обработки. Функция ShowWindowA помещает в очередь сообщений приложения синхронное сообщение WM_PAINT. Все синхронные сообщения попадают в очередь сообщений приложения и обрабатываются в порядке очередности. Это и послужило причиной того, что текст, выведенный в окно функцией TextOut, не был отображен в окне немедленно. В отличие от предыдущих двух доводов, данный вывод сделать довольно трудно, не имея дополнительных средств исследования, о которых мы говорили ранее. Косвенно это можно выяснить, закомментировав в программе вызов функции UpdateWindowA, основным назначением которой является посылка асинхронного сообщения WM_PAINT в оконную функцию. Асинхронные сообщения не ставятся в общую очередь приложения и сразу передаются в оконную функцию. Функция UpdateWindowA как раз и предназначена для таких ситуаций, в которых необходимо обновить содержимое окна. К этому вопросу, а так-

же к синхронным и асинхронным сообщениям мы еще вернемся, поэтому примите пока все сказанное здесь на веру.

Переместите теперь строки 189-205 (при этом именно переместите их, а не просто скопируйте) из функции WindowProc, разместив их сразу за вызовом функции ShowWindowA (перед UpdateWindowA). Перетранслируйте заново исходный текст приложения, получите новый вариант исполняемого модуля и запустите его. Поведение программы несколько изменилось. Файл create.wav воспроизведен, как обычно — после создания окна. Далее последовательно появились окно приложения и долгожданное сообщение в его области. И лишь после этого был воспроизведен звуковой файл paint.wav. Поведение приложения для этого варианта размещения фрагментов кода приводит к новым выводам.

- Функции Win32 API, выводящие что-либо в окно, должны размещаться после функций, реализующих действия по отображению окна.
- Приложение должно самостоятельно следить за актуальностью содержимого своего окна.
- Весь вывод на экран должен производиться в оконной функции.

Подтвердим правильность этих выводов следующими рассуждениями и экспериментами. Сверните и вновь разверните окно приложения. Вы увидите, что область окна вновь стала пустой, но при этом был воспроизведен звуковой файл create.wav. Исходя из того, что строки кода, воспроизводящие этот файл, находятся в оконной функции в том месте, где обрабатывается сообщение WM_PAINT, приходим к выводу — в очереди сообщений вновь оказалось это сообщение, и именно оно было обработано оконной функцией. Как сообщение WM_PAINT оказалось в очереди, если наше приложение на этот раз его туда не помещало, а сообщения WM_PAINT от ShowWindowA и UpdateWindowA к этому моменту уже обработаны и управление передано циклу обработки сообщений? Ответ напрашивается сам собой: это делает сама система Windows, именно она рассылает в очереди сообщений всех приложений, окна которых отображены на экране, сообщение WM_PAINT. Но при этом Windows не берет на себя обязанность хранить содержимое этих окон. За актуальность содержимого окна должно отвечать само приложение. Сообщение WM_PAINT служит для приложения сигналом обновить либо заново восстановить содержимое своего окна. Следовательно, если мы хотим, чтобы результаты предыдущих выводов на экран были актуальны, необходимо после получения сообщения WM_PAINT перерисовать содержимое окна приложения. Последнее рассуждение доказывает тезис о том, что весь вывод на экран должен производиться в оконной функции как реакция на поступление сообщения WM_PAINT.

Эти рассуждения показывают роль сообщений (и не только WM_PAINT), циркулирующих в системе. Теперь, понимая всю их важность, можно приступить к рассмотрению следующей важной части оконного Windows-приложения. При этом мы продолжим манипулировать фрагментами нашего программного кода (на этот раз звуковыми) для пояснения некоторых характерных моментов.

Цикл обработки сообщений

Сообщение в Win32 представляет собой объект особой структуры, формируемый Windows. Формирование и доставка этого объекта в нужное место в системе по-

зволяют управлять работой как самой системы Windows, так и загруженных Windows-приложений. Инициировать формирование сообщения могут несколько источников: пользователь, само приложение, система Windows, другие приложения. Именно наличие механизма сообщений позволяет Windows реализовать многозадачность, которая при работе на одном процессоре является, конечно же, псевдомультизадачностью. Windows поддерживает очередь сообщений для каждого приложения. Запуск приложения автоматически подразумевает формирование для него собственной очереди сообщений, даже если это приложение и не будет ею пользоваться. Последнее маловероятно, так как в этом случае у приложения не окажется связи с внешним миром и оно превратится в «вещь в себе».

Формат всех сообщений Windows одинаков и описывается структурой, шаблон которой содержится в файле winuser.h:

```
/*
 * Message structure
 */
typedef struct tagMSG {
 HWND   hwnd;
 UINT   message;
 WPARAM wParam;
 LPARAM lParam;
 DWORD  time;
 POINT  pt;
} MSG, *PMSG, NEAR *NPMSG, FAR *LPMSG;
```

В этой структуре все типы данных вам знакомы, за исключением одного — POINT. Этот тип данных описан во включаемом файле windef.h и представляет собой структуру вида

```
typedef struct tagPOINT
{  LONG  x;
 LONG y;} POINT, *PPOINT, NEAR *NPPOINT, FAR *LPPOINT;
```

На основе этого описания в файл windowsA.h помещено эквивалентное описание этой структуры в соответствии с синтаксисом ассемблера:

```
POINT   struc
x   ULONG   0
y   ULONG   0
    ends
MSG struc
meshwnd HWiND    0
mes     UINT?
wParam  UINT?
IParam  UINT?
time    dd  0
POINT   struc   {}
    ends
```

Поле meshwnd структуры MSG содержит значение дескриптора окна, которому предназначено сообщение. Это тот самый дескриптор, который возвращается функцией CreateWindowExA и, соответственно, однозначно идентифицирует окно в системе. Не забывайте, что приложение обычно имеет несколько окон, поэтому значение в поле meshwnd помогает приложению идентифицировать нужное окно.

В поле mes Windows помещает 32-разрядную константу — идентификатор сообщения, однозначно идентифицирующий тип сообщения. Для удобства все эти константы имеют символические имена, начинающиеся с префикса WM_ (Window Message). Посмотрите, каким образом они определены в файле winuser.h. Во включа-

емом файле для программ на ассемблере win32.inc (и нашем варианте этого файла — windowsA.h) тоже содержатся некоторые из этих констант. Если в ходе работы вам понадобятся отсутствующие константы, возьмите их из включаемых файлов компилятора C/C++, исправив описание в соответствии с синтаксисом ассемблера. В программе на языке C/C++ эти константы используются в оконной функции оператором switch для принятия решения о том, какая из его ветвей будет исполняться. В оконной функции каркасного Windows-приложения на ассемблере (строки 163-232) этот оператор моделируется командами условного и безусловного переходов (строки 168-174), а также командой cmp, в качестве второго операнда которой и выступает константа, обозначающая определенный тип сообщения.

Поля lParam и wParam предназначены для того, чтобы система Windows могла разместить в них дополнительную информацию о сообщении, необходимую для его правильной обработки. Эти поля, например, используются при обработке сообщений о выборе пунктов меню или о нажатии клавиш.

В поле time Windows записывает информацию о времени, когда сообщение было помещено в очередь сообщений.

И наконец, поле POINT содержит координаты указателя мыши в момент помещения сообщения в очередь.

Итак, представим, что в системе произошло какое-то событие, например, некоторому приложению необходимо перерисовать свое окно, в результате Чего система Windows сформировала сообщение WM_PAINT. Данное сообщение попадает в очередь сообщений приложения, создавшего окно. Для того чтобы приложение могло обработать это (или любое другое) сообщение, ему необходимо сначала его обнаружить в очереди сообщений. С этой целью после отображения окна на экране программа «входит» в специальный цикл, называемый *циклом обработки сообщений* (строки 139–156). Выйти из этого цикла можно только по приходу сообщения WM_QUIT. В этом случае функция GetMessageA возвращает нулевое значение, и команда условного перехода в строке 146 передает управление на конец цикла обработки сообщений. В случае прихода других сообщений функция GetMessageA возвращает ненулевое значение, в результате чего и осуществляется вход непосредственно в тело цикла обработки сообщения. Функции GetMessageA передается несколько параметров (строки 140-143):

- message — указатель на экземпляр структуры MSG (строка 42). Во время работы функция GetMessageA извлекает сообщение из очереди сообщений приложения и на основе информации в нем инициализирует поля экземпляра структуры message. Таким образом, приложение получает полный доступ ко всей информации в сообщении, сформированном Windows;
- hWnd — в поле передается дескриптор окна, сообщения для которого должны будут выбираться функцией GetMessageA. Параметр позволяет создать своеобразный фильтр, заставляющий функцию GetMessageA выбирать из очереди и передавать в цикл обработки сообщений сообщения лишь для определенного окна данного приложения. Если hWnd = NULL, то GetMessageA будет выбирать из очереди сообщения для всех окон;

- wMsgFilterMin и wMsgFilterMax — значения данных параметров также позволяют создавать фильтр для выбираемых функцией GetMessageA сообщений. Они задают диапазон номеров сообщений (поле mes структуры MSG), которые будут выбираться из очереди функцией GetMessageA и передаваться в цикл обработки сообщений. Параметр wMsgFilterMin задает минимальное значение параметра mes, а параметр wMsgFilterMax, соответственно, максимальное значение.

Функция GetMessageA выполняет следующие действия.

1. Постоянно просматривает очередь сообщений.
2. Выбирает сообщения, удовлетворяющие заданным в функции параметрам.
3. Заносит информацию о сообщении в экземпляр структуры MSG (строка 42).
4. Передает управление в цикл обработки сообщений.

Цикл обработки сообщений (строки 147-155) состоит всего из двух функций: TranslateMessage и DispatchMessageA. Эти функции имеют единственный параметр — указатель на экземпляр структуры MSG, предварительно заполненный информацией о сообщении функцией GetMessageA.

Функция TranslateMessage предназначена для обнаружения сообщений от клавиатуры для данного приложения. Если приложение самостоятельно не обрабатывает ввод с клавиатуры, то эти сообщения передаются для обработки обратно Windows.

Функция DispatchMessageA предназначена для передачи сообщения оконной функции. Такая передача производится не напрямую, так как сама DispatchMessageA ничего не знает о месторасположении оконной функции, а косвенно — посредством системы Windows. Это делается следующим образом:

1. Функция DispatchMessageA возвращает сообщение операционной системе.
2. Windows, используя описание класса окна, передает сообщение нужной оконной функции приложения.
3. После обработки сообщения оконной функцией управление возвращается операционной системе.
4. Windows передает управление функции DispatchMessageA.
5. DispatchMessageA завершает свое выполнение.

Так как вызов функции DispatchMessageA является последним в цикле, то управление опять передается функции GetMessageA, которая выбирает очередное сообщение из очереди сообщений и, если оно удовлетворяет параметрам, заданным при вызове функции, выполняет тело цикла. Цикл обработки сообщений выполняется до тех пор, пока не приходит сообщение WM_QUIT. Получение этого сообщения — единственное условие, при котором программа может выйти из цикла обработки сообщений.

Завершение выполнения приложения

Выход из цикла обработки сообщений означает одно — необходимо завершить программу. В программе на C/C++ для этого непосредственно за циклом обработки сообщений помещается оператор return (см. листинг 16.1):

```
return  lpMsg.wParam;
```

В качестве операнда в операторе return используется значение поля wParam экземпляра структуры MSG — lpMsg. Значение этого поля формируется значением соответствующего поля последнего сообщения, выбранного функцией GetMessageA из очереди. Нетрудно догадаться, что этим сообщением было WM_QUIT.

Листинг 16.3 позволяет посмотреть, каким образом процесс завершения Windows-приложения реализован компилятором языка C/C++:

```
004012C6    call _WinMain@16
004012CB    push eax
004012CC    call _exit
...
00401380_exit  proc near   ; CODE  XREF: start+A9_p
...
00401389    call sub_4013C0  ;_doexit
...
00401391_exit   endp
...
004013C0 ;_doexit
...
004013D1   call ds:GetCurrentProcess
004013D7    pusheax
004013D8   call ds:TerminateProcess
...
00401458   pushesi
00401459   call ds:ExitProcess
...
00401462    retn
```

Из полного варианта листинга 16.3 видно, что процедура WinMain возвращает в регистре eax значение wParam сообщения WM_QUIT. Затем вызывается локальная процедура _exit, предназначенная для выполнения определенных действий по завершению приложения. Процедура_exit, в свою очередь, вызывает другую локальную процедуру — _doexit. Ее текст представляет наибольший интерес для нас, так как в нем мы видим те функции Win32 API, которые непосредственно выполняют работу по удалению приложения из системы Windows, — это три функции: GetCurrentProcess, TerminateProcess и ExitProcess. Для завершения работы приложения достаточно использовать только функцию ExitProcess, что и сделано в разработанной нами программе (строки 158-161 листинга 16.4).

На этом рассмотрение работы главной функции стандартного Windows-приложения можно считать оконченным. Видимо, вы обратили внимание на то, что до сих пор вся работа шла в интересах Windows: инициализировались определенные структуры данных, вызывались строго определенные функции и т. д. А где же полезная работа приложения? Выполнением этой работы занимается оконная функция. Если быть более точным, то она выступает координатором этой работы. Далее разберемся с тем, как реализовать оконную функцию Windows-приложения на языке ассемблера.

Обработка сообщений в оконной функции

Оконная функция призвана организовать адекватную реакцию со стороны Windows-приложения на действия пользователя и поддерживать в актуальном состоянии то окно приложения, сообщения которого она обрабатывает. Приложение может иметь несколько оконных функций. Их количество определяется количеством классов окон, зарегистрированных в системе функцией RegisterClass(Ex). Если

вы помните, функции RegisterClass(Ex) посредством экземпляра структуры WNDCLASS передается указатель на определенную оконную функцию Windows-приложения. Данная функция до конца работы приложения связана с экземплярами окон, которые, в свою очередь, создаются другой функцией API — CreateWindowEX.

Когда для окна Windows-приложения появляется сообщение, операционная система Windows производит вызов соответствующей оконной функции. Ранее мы для упрощения говорили, что единственный источник появления сообщений — очередь сообщений приложения, но это не совсем так. Дело в том, что сообщения, в зависимости от источника их появления в оконной функции, могут быть двух типов: синхронные и асинхронные. К *синхронным* сообщениям относятся те сообщения, которые помещаются в очередь сообщений приложения и терпеливо ждут момента, когда они будут выбраны функцией GetMessage. После этого поступившие сообщения попадают в оконную функцию, где и производится их обработка. *Асинхронные* сообщения подобно пожарной машине попадают в оконную функцию в экстренном порядке, минуя все очереди. Асинхронные сообщения, в частности, инициируются некоторыми функциями Win32 API, такими как CreateWindow(Ex) или UpdateWIndow. Координацию синхронных и асинхронных сообщений осуществляет Windows. Если рассматривать синхронное сообщение, то его извлечение производится функцией GetMessage с последующей передачей обратно в Windows функцией DispatchMessage. Асинхронное сообщение, независимо от источника, который инициирует его появление, сначала попадает в Windows и затем — в нужную оконную функцию.

Для чего реализуется именно такая схема, неужели функции DispatchMessage нельзя сразу передать сообщение в оконную функцию? Если бы это было так, в системе появилось бы единственное приложение-монополист, которое захватило бы все процессорное время своим циклом обработки сообщений. В схеме, реализованной в Windows, обработка сообщений приложением проводится в два этапа: на первом этапе приложение выбирает сообщение из очереди и отправляет его обратно во внутренние структуры Windows; на втором этапе Windows вызывает нужную оконную функцию приложения, передавая ей параметры сообщения. Преимущество этой схемы в том, что Windows самостоятельно решает все вопросы организации эффективной работы приложений.

Таким образом, при поступлении сообщения Windows вызывает оконную функцию и передает ей ряд параметров. Все они берутся из соответствующих полей сообщения. В нотации языка C/C++ заголовок оконной функции описан следующим образом (см. листинг 16.1):

```
LRESULT CALLBACK WindowProc (HWND hWnd, UINT message, WPARAM wParam,
  LPARAH lParam)
```

Здесь hWnd — дескриптор окна, которому предназначено сообщение; message — идентификатор сообщения, характеризующий тип сообщения; wParam и IParam — дополнительные параметры, являющиеся копиями соответствующих полей структуры поступившего сообщения.

Оставшиеся два поля структуры MSG: time и POINT — используются довольно редко, и при необходимости их значения можно извлечь непосредственно из экземпляра структуры сообщения.

В нашем примере Windows-приложения строки 164-167 обозначают начало оконной функции. Параметры этой функции Windows помещает в стек. Чтобы можно было работать с ними, используя символические имена, после заголовка функции поместим директиву ARG (см. главу 15).

Windows требует, чтобы оконная функция сохраняла значения регистров EBI, EDI и ESI. Причина — оконная функция должна быть рекурсивной. Например, возможна ситуация, когда несколько классов окон применяют одну и ту же оконную функцию для обработки сообщений, поступающих в созданные на базе этих классов окна. Имеет смысл сохранять не только названные регистры, используемые самой системой Windows, но и другие (за исключением EAX), если они задействованы в оконной функции. Исходя из требования рекурсивности, все переменные, используемые в оконной функции, должны быть локальными. Чтобы удовлетворить требование сохранения регистров, лучше применить (строка 166 листинга 16.4) соответствующее средство транслятора ассемблера — директиву USES (см. также главы 10 и 15). При этом транслятор вставляет в начало и конец оконной функции соответствующую последовательность команд ассемблера PUSH. Вы можете убедиться в этом, посмотрев файл листинга (файл .lst) программы из листинга 16.4 после ее трансляции.

Центральным местом оконной функции является синтаксическая конструкция, в задачу которой входит распознавание поступившего сообщения по его типу (параметр message) и передача управления на ту ветвь кода оконной функции, которая продолжает далее работу с параметрами сообщения. В языке C/C++ (см. листинг 16.1) для этого используется оператор switch (переключатель). В языке ассемблера такого средства нет, поэтому его приходиться моделировать (см. главу 11). В листинге 16.4 строки 168-174 показывают, как это можно сделать с помощью команды CMP, а также команд условного (JE) и безусловного (JMP) переходов.

Соответствие символических имен константам, обозначающим тип сообщений Windows, приведено во включаемом файле windowsA.h. Совершенно необязательно анализировать типы всех возможных сообщений, параметры которых передаются в оконную функцию. Структуру, подобную строкам 168–174, и вообще структуру всей оконной функции можно сравнить с ситом, а сообщения — с зернами разной величины. Отверстия в сите непростые — каждое из них пропускает зерно своего размера. Зерна (сообщения) в сите (оконной функции) не задерживаются, попав в него, они через одно из отверстий обязательно уходят. Уходят куда? Это опять-таки определяется особой структурой нашего сита. Представьте, что к каждому отверстию припаяна трубка, которая определяет, куда данное зернышко попадет для дальнейшей обработки. Таким образом, каждому сообщению, попадающему в оконную функцию, соответствует ветвь в коде процедуры, которая начинает обработку этого сообщения. В ходе этой обработки, возможно, понадобится вызов других функций или применение синтаксических конструкций, подобных описанному ситу, для дальнейшей более тонкой селекции сообщений, но уже одного типа (по полям lParam или wParam). Для наглядности аналогию структуры оконной функции с ситом можно упрощенно представить в виде схемы (рис. 16.1).

Представленная схема показывает, что оконная функция, обрабатывающая сообщения, имеет одну точку входа и множество точек выхода. Выход осуществляется из той ветви оконной функции, где обрабатывалось сообщение. Сообщения,

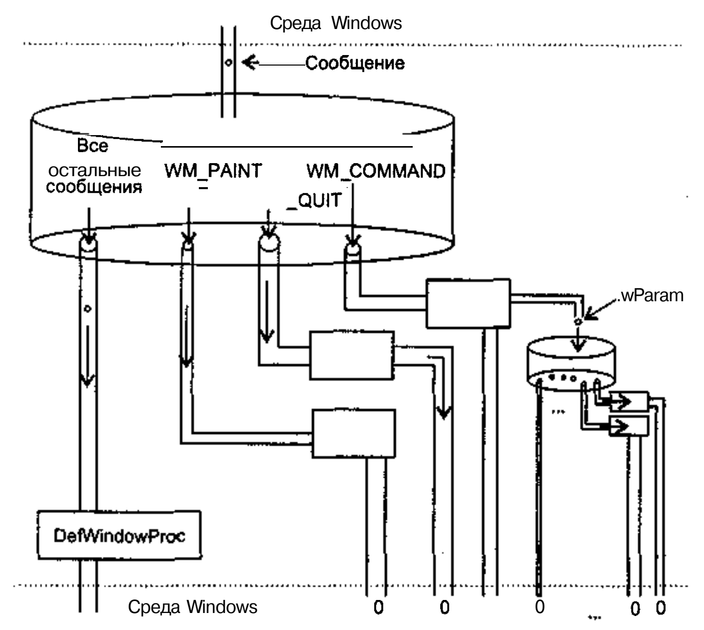


Рис. 16.1. Абстрактное представление структуры оконной функции Windows

для которых не предусмотрена отдельная обработка, должны обрабатываться функцией DefWindowProc (строки 219–227). Эта процедура по отношению к переданным ей сообщениям предпринимает действия по умолчанию. В принципе, ей можно передавать на обработку все сообщения, что в контексте рассмотренной ранее абстракции означает сито, содержащее одно отверстие, в которое проваливаются все поступающие сообщения.

По завершении работы оконная функция формирует значение в регистре EAX. Если сообщение обрабатывалось в оконной функции, то в EAX необходимо поместить нулевое значение. Если обработка осуществлялась по умолчанию, то есть функцией DefWindowProc, то в EAX уже сформировано возвращаемое значение, и именно его нужно возвратить в качестве результата работы оконной функции.

В нашем примере оконная функция обрабатывает три сообщения: создание окна WM_CREATE, перерисовку области окна WM_PAINT, закрытие окна WM_DESTROY.

Более подробно с условиями возникновения и обработкой этих и других сообщений можно познакомиться в литературе по Windows, где эти вопросы освещены более полно.

Средства TASM для разработки Windows-приложений

Ранее мы подробно разобрались с тем, что собой представляет простое Windows-приложение, написанное на языке ассемблера. Излагая материал, мы упоминали имена файлов, которые нужны для получения работоспособного исполняемого модуля программы. Для устранения возможных неясностей соберем и систематизируем эту информацию.

При разработке Windows-приложений на языке ассемблера с помощью Win32 API вам нужен один из пакетов ассемблера — не обойтись без пакета TASM версии 5.0. Современные 32-разрядные операционные системы Windows используют формат PE исполняемого файла. В состав пакета TASM 5.0 входят два компилятора ассемблера — 16- и 32-разрядный. Они имеют имена исполняемых файлов, соответственно, tasm.exe и tasm32.exe. То же касается и редакторов связей — tlink.exe и tlink32.exe. Получить файл формата PE можно только при совместном использовании файлов tasm32.exe и tlink32.exe. Для удобства работы, подобно тому, как мы это делали для 16-разрядных программ в главе 6, скопируйте все файлы 32-разрядной версии в отдельный каталог и назовите его, например, Work32. Сюда же необходимо поместить файлы windowsA.inc и import32.lib, находящиеся среди файлов, прилагаемых к книге.

Для создания программы нужны еще два файла: файл определений компоновщика и файл описания ресурсов. Файл описания ресурсов будет рассмотрен в следующем разделе. Что касается *файла определений компоновщика,* то его содержимое довольно подробно описывается в различных источниках, и его роль ничем не отличается от роли аналогичного файла при разработке Windows-приложений на других языках. Назначение файла определений компоновщика состоит в том, чтобы предоставить редактору связей информацию о способе загрузки программы. Несмотря на то что в архитектуре Win32 нет особого смысла использовать данный файл, редактор tlink32.exe требует указания этого файла среди файлов, подаваемых ему в качестве входных. Вы можете без проблем задействовать готовый вариант этого файла, находящийся среди файлов, прилагаемых к книге.

Перечислим необходимые для разработки Windows-приложения файлы.

- Файл с исходным текстом программы (.asm). Формируется программистом.
- Включаемый файл с описаниями структур данных и констант Win32 (.inc или .ash). Файл формируется программистом по мере расширения используемых им средств Win32. Источником информации для этого файла служат включаемые файлы (.h) из пакета компилятора C/C++, например VC++ версии 4.0 и выше.
- Файл с библиотекой импорта import32.lib. Этот файл требуется компоновщику для разрешения внешних ссылок на функции Win32 API. Вы можете создать этот файл сами. Такая необходимость может возникнуть, если вам понадобятся функции из библиотек DLL, информация о которых отсутствует в существующем варианте файла import32.lib. Для этого существует специальная утилита implib.exe, поставляемая в пакете TASM 5.O. Командная строка для ее запуска имеет вид

  ```
  implib имя_файла_lib   список_dll_библиотек
  ```

  Получить информацию о местонахождении конкретной функции Win32 API довольно просто. Многие справочные руководства по Windows при описании конкретной функции приводят и информацию о библиотеке DLL, где эта функция содержится.
- Файл с описанием ресурсов, используемых в приложении.
- Другие файлы. Например, в рассмотренной нами программе каркасного приложения (см. листинг 16.4) используются звуковые файлы (.wav).

- Файлы tasm32, tlink32.exe и, возможно, некоторые другие вспомогательные файлы из пакета TASM 5.0. Следите внимательно за сообщениями. В том случае, если какого-либо файла будет недоставать, его нужно просто найти в каталоге \bin пакета TASM 5.0 и скопировать в свой рабочий каталог. Непосредственно в каталоге \bin работать не рекомендуется, иначе он моментально превратится в слабоструктурируемое нагромождение файлов.
- Компилятор ресурсов brc32.exe или brcc32.exe. Компиляторы взяты из пакета C/C++ фирмы Borland. Но если вы работаете с пакетом VC++, то вам может понадобиться компилятор ресурсов, входящий в этот пакет. Он называется rc.exe.
- Файл makefile и утилита make.exe. Эти файлы призваны облегчить процесс сборки приложения в единый исполняемый модуль.

Приведенный список файлов, необходимых для сборки Windows-приложения, довольно велик. Ранее процесс получения исполняемого файла у нас был простым и вполне управлялся из командной строки (без использования, например, make-файлов). Более сложные приложения требуют учета взаимосвязей между несколькими файлами. Данные файлы, в свою очередь, создаются или обрабатываются разными программными средствами, которые иногда требуют задания режимов работы многочисленными параметрами. Запоминать их и постоянно вводить вручную тяжело, и такая работа вряд ли может быть признана эффективной. Для облегчения процесса получения исполняемого файла используйте возможности, предоставляемые make-файлами.

Make-файл для программиста — существенное облегчение в его работе. Тщательно разработав один раз make-файл для создания исполняемого файла своего приложения, вы впоследствии избавите себя от рутинной работы по формированию необходимых для этого командных строк. Второй положительный эффект от использования make-файлов — упрощается работа автора по описанию процесса получения исполняемого модуля. Кстати, в роли автора можете оказаться и вы, когда разработаете свою программу и вместо длинного описания процесса ее сборки предоставите пользователям make-файл, дополнив его необходимыми комментариями. Информацию о правилах написания make-файлов можно найти среди прилагаемых к книге файлов в каталоге к главе 6. Make-файл для сборки приложения (см. листинг 16.4) представлен в листинге 16.6.

Листинг 16.6. Пример make-файла для создания приложения prg16_1.exe

```
NAME = prg16_1
OBJS = $(NAME).obj
DEF = $(NAME).def
TASMDEBUG=/zi
LINKDEBUG=/v
IMPORT=import32
$(NAME).EXE: $(OBJS) $(DEF)
tlink32 /Tpe /aa /c $(LINKDEBUG) $(OBJS), $(NAME), , $(IMPORT), $(DEF)
.asm.obj:
tasm32 $(TASMDEBUG) /ml $&.asm, , ,
```

Поясним наиболее значимые элементы приведенного файла. Мы уже упоминали, что трансляция исходного файла производится программой tasm32.exe. При

вызове ей передается ряд ключей (строки 1-6). Для формирования отладочной информации вы можете указывать значения макрооператоров TASMDEBUG = /zi (строка 4) и LINKDEBUG = /v (строка 5). Кроме этих необязательных ключей в строке 10 присутствует обязательный ключ /ml, который требует, чтобы транслятор ассемблера различал строчные и прописные буквы в идентификаторах программы. Как вы смогли убедиться, это очень удобно при написании программ для Windows. Строка 8 содержит вызов компоновщика tlink32.exe. Описания других ключей для этой программы приведены в главе 6 и приложении В (http://www.piter.com/download).

При запуске программы make.exe не забывайте о нехитром, но полезном приеме с перенаправлением вывода экранных сообщений в файл с помощью символа «>»:

```
MAKE.EXE -DDEBUG > p
```

В файле p текущего каталога читайте диагностические сообщения о процессе построения приложения.

Углубленное программирование на ассемблере для Win32

Реализация описанного ранее процесса разработки простого Windows-приложения на языке ассемблера может отнять довольно много сил и времени у неподготовленного человека. Но, скорее всего, это не будет пустой тратой драгоценных для любого программиста жизненных сил. Цель материала, изложенного именно в таком виде, — демонстрация того, что разработка Windows-приложения на языке ассемблера — не такое уж нереальное дело. Напротив, у него даже есть свои достоинства. Нужно отметить, что объем учебного материала, необходимого для описания процесса разработки каркасного приложения для Windows, не зависит от языка, на котором предполагается вести программирование, так как основное внимание уделяется не столько средствам языка, сколько описанию требований к функционированию приложения со стороны Windows. Каркасное приложение является простейшей программой для Windows, которая в лучшем случае выводит строку текста в окно приложения. С точки зрения сложности, не имеет смысла даже проводить ее сравнение с аналогичной программой для MS-DOS, выводящей строку на экран. Логику работы и способы реализации такой программы для MS-DOS в худшем случае можно объяснить минут за десять. Для объяснения логики работы каркасного Windows-приложения неподготовленному слушателю придется прочитать целую лекцию, может быть, и не одну. И это простейшая программа. А где же предел? Какими минимальными знаниями и умениями должен обладать программист, чтобы утверждать, что он является, если, конечно, можно так выразиться, профессиональным Windows-программистом. Не претендуя на безусловную истину, попытаемся перечислить некоторые проблемы, которые программист должен научиться решать в первую очередь.

- Нужно понять общие принципы построения программы, работа которой управляется сообщениями.
- Нужно научиться выводить текст и графику в область окна приложения. Основная проблема здесь состоит в умении эффективно использовать совокуп-

ность средств Win32 API. Сам процесс формирования изображения в окне Windows напоминает процесс формирования изображения в видеобуфере, как это делалось в MS-DOS. Оба эти варианта вывода изображения можно сравнить с рисованием цветными мелками на школьной доске. Для того чтобы обновить содержимое окна, его необходимо либо полностью вывести заново, либо сначала удалить ненужные фрагменты, сформировать на их месте новые и затем вывести в определенное место в окне. Эта проблема называется проблемой *перерисовки изображения,* и она тесно связана с тем, насколько эффективно решается следующая проблема.

- Нужно организовать адекватную обработку сообщений. Эффективность и правильность работы программы напрямую зависит от того, насколько правильно в ней организована обработка сообщений. В самом начале процесса обучения написанию программ для Windows вы столкнетесь с необходимостью обработки такого сообщения, как WM_PAINT. Проблема здесь заключается в том, что Windows не сохраняет содержимое окна или части окна при его свертывании или скрытии под другим окном. Следить за содержимым своих окон должно само приложение, а точнее, соответствующая оконная функция. Более подробно о решении этой проблемы мы узнаем при рассмотрении вопроса перерисовки изображения.
- Нужно научиться создавать интерфейсную часть приложения. Интерфейс приложения — это его визитная карточка. Первым, на что обращает внимание пользователь, тем более если он непрофессионал, является именно этот элемент работы приложения. Более того, движущей силой развития самой системы Windows является стремление к реализации идеи идеального интерфейса. На сегодняшний день эту роль играет оконный интерфейс. Что будет завтра, пока не ясно, так как оконный интерфейс действительно решает многие проблемы и до исчерпания его потребительского ресурса, наверное, еще далеко. Основа оконного интерфейса — окно, в котором имеются две области: управляющая, с ее помощью осуществляется управление работой окна, и пользовательская, которая обычно занимает большую часть окна, и именно в ней пользователь формирует некоторое изображение. Управляющая область окна приложения состоит из более элементарных интерфейсных компонентов: меню, окон диалога, кнопок, панелей и т. д. Создание и организация работы со многими из этих компонентов поддерживается Windows с помощью функций Win32 API.
- Нужно научиться обрабатывать пользовательский ввод.

Каждый пункт этого списка представляет лишь вершину некоторой иерархии более частных проблем и может быть довольно глубоко детализирован вглубь. Конечно, в рамках нашего изложения сделать это не представляется возможным, да и вряд ли нужно. Подобные вопросы хорошо изложены в литературе. Материал данной главы подобран так, чтобы показать, как отражается на процессе разработки Windows-приложения выбор ассемблера в качестве основного языка программирования. Поэтому основное внимание мы уделяем не тому, как реализовать те или иные элементы пользовательского интерфейса, а технологии сборки работо-

способного приложения с помощью пакета TASM (для MASM отличие только в инструментарии). Исходя из этого далее покажем, как использовать ресурсы в Windows-приложениях, написанных на ассемблере.

Ресурсы Windows-приложений на языке ассемблера

Для включения ресурсов в Windows-приложения, написанные на ассемблере, задействуется та же самая технология, что и для программ на языках C/C++. *Ресурс* — это специальный объект, используемый программой, но не определяемый в ее теле. К ресурсам относятся следующие элементы пользовательского интерфейса: значки, меню, окна диалога, растровые изображения.

Определение ресурсов производится в текстовом файле с расширением .rc, в котором для описания каждого типа ресурса используются специальные операторы. Подготовку этого файла можно вести двумя способами: ручным и автоматизированным.

Ручной способ предполагает:

- что разработчик ресурса хорошо знает операторы, необходимые для описания конкретного ресурса;
- что ввод текста ресурсного файла выполняется с помощью редактора, который не добавляет в текст элементы форматирования, например редактора Блокнот (notepad.exe), входящего в состав программного обеспечения Windows.

Автоматический способ создания ресурсного файла предполагает использование специальной программы — редактора ресурсов, который позволяет визуализировать процесс создания ресурса. Конечные результаты работы этой программы могут быть двух видов: в виде текстового файла с расширением .rc, который впоследствии можно подвергнуть ручному редактированию, либо в виде двоичного файла, уже пригодного к включению в исполняемый файл приложения.

Будем предполагать, что описание ресурсов в текстовом виде получено и находится в файле ресурсов с расширением .rc. Далее это описание должно быть преобразовано в вид, пригодный для включения в общий исполняемый файл приложения. Для этого необходимо выполнить перечисленные далее шаги.

1. Откомпилировать ресурсный файл. На этом шаге выполняется преобразование текстового представления ресурсного файла с расширением .rc в двоичное представление с расширением .res. Для этого в пакете TASM есть специальная программа brc32.exe — компилятор ресурсов.
2. Включить ресурсы в исполняемый файл приложения. Это действие выполняет компоновщик tlink32.exe, которому в качестве последнего параметра должно быть передано имя ресурсного файла (.res).

Далее на конкретных примерах мы рассмотрим порядок применения этих программных средств для включения некоторых типов ресурсов в Windows-приложения.

Меню в Windows-приложениях

Меню в системе Windows являются, пожалуй, самым распространенным элементом пользовательского интерфейса. Мы не будем особенно вдаваться в детали раз-

работки приложения с меню, так как это уже делалось при рассмотрении каркасного приложения. Содержание этого процесса стандартное, поэтому мы остановимся только на специфических моментах его реализации при разработке приложения на ассемблере. Для того чтобы включить меню в приложение, необходимо реализовать следующую последовательность шагов.

1. Разработка сценария меню. Перед тем как приступить к процессу включения меню в конкретное приложение, разработаем его логическую схему. Этот шаг необходим для того, чтобы уже на стадии проектирования обеспечить эргономические свойства приложения. Ведь меню — это один из немногих элементов интерфейса, с которым пользователь вашего приложения будет постоянно иметь дело. Поэтому схема меню должна иметь наглядную иерархическую структуру с логически увязанными между собой пунктами, что поможет пользователю эффективно задействовать все возможности приложения. Для того чтобы вести предметный разговор, поставим себе задачу разработать для окна нашего приложения главное меню. При этом мы исследуем возможности вывода в окно приложения текста и графики, а также покажем способы решения общих проблем, связанных с разработкой приложения. Наше меню будет довольно простым и состоять из трех пунктов: Текст, Графика и 0 приложении, — причем первые два пункта будут открывать доступ к подменю. Иерархическая структура меню представлена на рис. 16.2.

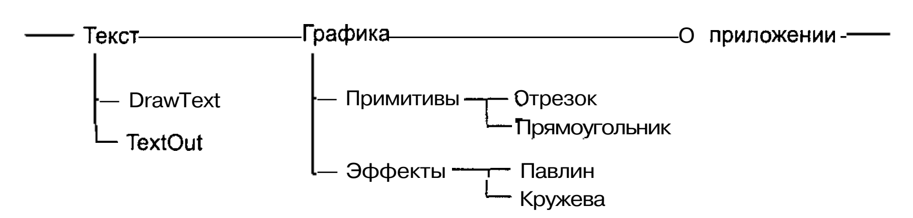


Рис. 16.2. Иерархическая структура меню

2. Описание схемы меню в файле ресурсов. Для выполнения этого этапа требуются специальные операторы. В нашем случае файл ресурсов будет выглядеть следующим образом:

```
//Текст файла menu.rc
#include"menu.h"
MYMENU MENU DISCARDABLE
{
POPUP "&Текст"
    {
    MENUITEM "&DrawText",   IDM_DRAWTEXT
    MENUITEM "&TextOut",IDM_TEXTOUT
    }
POPUP "&Графика"
    {
    POPUP "&Примитивы"
        {
            MENUITEM "&Отрезок",IDM_LENGTH
            MENUITEM "&Прямоугольник",  IDM_RECTANGLE
        }
    POPUP "&Эффекты"
        {
```

```
            MENUITEM "&Павлин", IDM_PEACOCK
            MENUITEM "&Кружева",IDM_LACES
        }
    }
MENUITEM "&About", IDM_ABOUT
}
```

3. Составление текста включаемого файла, необходимого для компиляции ресурсного файла. В нашем случае файл называется menu.h и выглядит следующим образом:

```
#define MYMENU 999
#define IDM_DRAWTEXT100
#define IDM_TEXTOUT 101
#define IDM_LENGTH 102
#define IDM_RECTANGLE   103
#define IDM_PEACOCK 104
#define IDM_LACES       105
#define IDM_ABOUT       106
```

   В этом файле идентификаторам пунктов меню назначаются константы, которые впоследствии будут передаваться в оконную процедуру в младшем слове параметра wParam сообщения WM_COMMAND. Заметьте, что имени самого меню также назначена константа. К этому моменту мы вернемся чуть позже.

4. Компиляция ресурсного файла. Для этого используется утилита brc32.exe:

```
brc32 [ключи ...] menu.rc
```

   Если утилита заканчивает свою работу нормально, то создается файл с расширением .res (menu.res). В случае, если утилита обнаруживает ошибки в исходном ресурсном файле, она выдает на экран соответствующие диагностические сообщения. Для удобства работы с ними их можно записать в файл, перенаправив вывод сообщений с экрана в файл, используя оператор командной строки «>»:

```
brc32 [ключи ...] имя_файла.rc > Имя_Файла_Диагностики
```

   например, так:

```
brc32 menu.rc > p
```

5. Подключение меню на стадии регистрации того окна приложения, для работы с которым оно будет использоваться. Для этого существуют два основных способа:

   - первый способ — поместить в поле lpszMenuName экземпляра структуры WNDCLASS (строка 101 листинга 16.4) указатель на поле, содержащее имя меню:

```
.data
...
menu db "MYMENU"
...
.code
...
    mov dword ptr wcl.lpszMenuName, offset menu
...
```

   - второй способ — назначить ресурсу меню символическую константу, которая расположена в поле имени оператора MENU файла ресурсов:

```
56 MENUDISCARDABLE ;оператор из файла ресурсов
```

Чтобы использовать упомянутую на последнем шаге символическую константу, необходимо вставить в файлы, из которых собирается приложение, дополнительные строки:

- Файл ресурсов:

```
MYMENU  MENUDISCARDABLE
```

- Файл menu.h:

```
#define MYMENU     56
```

- Файл menu.inc:

```
MYMENU   equ  56
```

- Строка 101 из листинга 16.4 должна выглядеть так:

```
mov dword ptr wcl.lpszMenuName, MYMENU
```

Или так:

```
mov dword ptr wcl.lpszMenuName, 56
```

Вторым способом мы фактически смоделировали макрос MAKEINTRESOURCE, известный Windows-программистам, пишущим приложения на языке C/C++.

После внесения всех изменений в верхней части окна появится строка меню. Далее необходимо в оконную функцию включить команды, реализующие реакцию на выбор пунктов этого меню. Эта информация передается в младшем слове поля wParam сообщения WM_COMMAND. В листинге 16.7 приведен измененный текст каркасного приложения, дополненный меню.

Листинг 16.7. Пример приложения с использованием меню

```
 ;Пример приложения для Win32 с использованием меню
.386
 locals ;разрешает применение локальных меток в программе
 .model flat, STDCALL   ;модель памяти flat,
 ;STDCALL - передача параметров в стиле С (справа налево),
 ;вызываемая процедура чистит за собой стек
 include windowA.inc;включаемый файл с описаниями базовых структур
                        ;и констант Win32
 include menu.inc       ;включаемый файл с определением имен
                        ;идентификатор меню
;Объявление внешними используемых в данной программе
;функций Win32 (ASCII):
    extrn   GetModuleHandleA:PROC
<...>
    extrn   GetClientRect:PROC
     ;объявление оконной функции объектом,
     ;видимым за пределами данного кода
    public  WindowProc
     .data
    hwnd    dd  0
    hInst   dd  0
    ;lpVersionInformation  OSVERSIONINFO  <?>
    wcl WNDCLASSEX  <?>
    message MSG <?>
    ps  PAINTSTRUCT <?>
    lpRect  RECT<?>
    szClassName db  'Приложение Win32', 0
    szTitleName db  'Каркасное приложение Win32 на ассемблере', 0
    MesWindow   db  'Привет! Ну как вам процесс разработки приложения
                    на ассемблере?'
    MesWindowLen=   $-MesWindow
```

продолжение ⇨

Листинг 16.7 *(продолжение)*

```
playFileCreate  db  'create.wav', 0
playFilePaint   db  'paint.wav', 0
playFileDestroy db  'destroy.wav', 0
.code
start   proc near
;точка входа в программу:
;начало стартового кода
...
;строки 55-72 листинга 16.4
...
;конец стартового кода
WinMain:
...
;строки 75—100 листинга 16.4
...
   mov dword   ptr wcl.lpszMenuName, MYMENU
;строки 102-149 листинга 16.4
...
start   endp
;-----.........--.....WindowProc------....-----....---.....---
WindowProc  proc
arg @@hwnd:DWORD, @@mes:DWORD, @@wparam:DWORD, @@lparam:DWORD
uses ebx, edi, esi, ebx ;эти регистры обязательно должны сохраняться
local   @@hdc:DWORD
    cmp @@mes, WM_DESTROY
    je  wmdestroy
    cmp @@mes, WM_CREATE
    je  wmcreate
    cmp @@mes, WM_PAINT
    je  wmpaint
    cmp @@mes, WM_COMMAND
    je  wmcommand
    jmp default
wmcreate:
...
;строки 176-212 листинга 16.4
...
;послать сообщение WM_QUIT
;готовим вызов VOID PostQuitMessage(int nExitCode)
    push    0
    call    PostQuitMessage
    mov     eax, 0  ;возвращаемое значение - 0
    jmp     exit_wndproc
wmcommand:
;вызов процедуры обработки сообщений от меню
;MenuProc (DWORD @@hwnd, DWORD @@wparam)
    push    @@wparam
    push    @@hwnd
    call    MenuProc
    jmp     exit_wndproc
default:
;обработка по умолчанию
;готовим вызов LRESULT DefWindowProc (HWND hWnd  UINT Msg,
;WPARAM wParam, LPARAM lParam)
    push    @@lparam
    push    @@wparam
    push    @@mes
    push    @@hwnd
    call    DefWindowProcA
    jmp     exit_wndproc
;...
exit_wndproc:
    ret
```

```
WindowProc  endp
;--------------------MenuProc--------------------------------
;обработка сообщений от меню
MenuProcproc
arg     @@hwnd:DWORD, @@wparam:DWORD
uses    ebx
local   @@hdc:DWORD
     mov ebx, @@wparam   ;в bx идентификатор меню
     cmp bx, IDM_DRAWTEXT
     je  @@idmdrawtext
     cmp bx, IDM_TEXTOUT
     je  @@idmtextout
     cmp bx, IDM_LENGTH
     je  @@idmlength
     cmp bx, IDM_RECTANGLE
     je  @@idmrectangle
     cmp bx, IDM_PEACOCK
     je  @@idmpeacock
     cmp bx, IDM_LACES
     je  @@idmlaces
     cmp bx, IDM_ABOUT
     je  @@idmabout
     jmp @@exit
@@idmdrawtext:
;получаем контекст устройства HDC GetDC(HWND hWnd)
     push    @@hwnd
     call    GetDC
     mov     @@hdc, eax
;получаем размер рабочей области
;BOOL GetClientRect(HWND hWnd, LPRECT IpRect)
     push    offset IpRect
     push    @@hwnd
     call    GetClientRect
;выводим строку текста в окно
;int DrawText(HDC hDC, LPCTSTR IpString, int nCount,
;LPRECT IpRect, UINT uFormat)
     push    DT_SINGLELINE+DT_BOTTOM
     push    offset IpRect
     push    -1
     push    offset @@TXT_DRAWTEXT
     push    @@hdc
     call    DrawTextA
;освобождаем контекст int ReleaseDC(HWND hWnd, HDC hDC)
     push    @@hdc
     push    @@hwnd
     call    ReleaseDC
     jmp     @@exit
@@idmtextout:
     push    @@hwnd
     call    GetDC    ;получаем контекст устройства
     mov     @@hdc, eax
;выводим строку текста в окно BOOL TextOut(HDC hdc, int nXStart,
;int nYStart, LPCTSTR IpString, int cbString)
     push    lenTXT_TEXTOUT
     push    offset @@TXT_TEXTOUT
     push    150
     push    10
     push    @@hdc
     call    TextOutA
     push    @@hdc
     push    @@hwnd
     call    ReleaseDC    освобождаем контекст устройства
     jmp     @@exit
@@idmlength:
     push    MB_ICONINFORMATION+MB_OK
```

продолжение ➩

Листинг 16.7 *(продолжение)*

```
<177>       push    offset szTitleName
<178>       push    offset @@TXT_LENGTH
<179>       push    @@hwnd
<180>       call    MessageBoxA
<181>       jmp     @@exit
<182>   @@idmrectangle:
<183>       push    MB_ICONINFORMATION+MB_OK
<184>       push    offset szTitleName
<185>       push    offset @@TXT_RECTANGLE
<186>       push    @@hwnd
<187>       call    MessageBoxA
<188>       jmp     @@exit
<189>   @@idmpeacock:
<190>       push    MB_ICONINFORMATION+MB_OK
<191>       push    offset szTitleName
<192>       push    offset @@TXT_PEACOCK
<193>       push    @@hwnd
<194>       call    MessageBoxA
<195>       jmp     @@exit
<196>   @@idmlaces:
<197>       push    MB_ICONINFORMATION+MB_OK
<198>       push    offset szTitleName
<199>       push    offset @@TXT_LACES
<200>       push    @@hwnd
<201>       call    MessageBoxA
<202>       jmp     @@exit
<203>   @@idmabout:
<204>       push    MB_ICONINFORMATION+MB_OK
<205>       push    offset szTitleName
<206>       push    offset @@TXT_ABOUT
<207>       push    @@hwnd
<208>       call    MessageBoxA
<209>       jmp     @@exit
<210>   ;...
<211>   @@exit:
<212>       mov eax, 0
<213>       ret
<214>   @@TXT_ABOUT db  'IDM_ABOUT', 0
<215>   @@TXT_LACES db  'IDM_LACES', 0
<216>   @@TXT_PEACOCK   db  'IDM_PEACOCK', 0
<217>   @@TXT_RECTANGLE db  'IDM_RECTANGLE', 0
<218>   @@TXT_LENGTHdb  'IDM_LENGTH', 0
<219>   @@TXT_TEXTOUT   db  'Текст выведен функцией TEXTOUT'
<220>   lenTXT_TEXTOUT=$-@@TXT_TEXTOUT
<221>   @@TXT_DRAWTEXT  db  'Текст выведен функцией DRAWTEXT', 0
<222>   MenuProcendp
<223>   end start
```

Строки 98-100 листинга 16.7 показывают, что для обработки сообщения WM_COMMAND вызывается процедура MenuProc. Подход, при котором программа на ассемблере структурируется на более мелкие части с использованием механизма процедур, при программировании для Windows особенно полезен и должен быть преобладающим. Для большей наглядности мы не задействуем макрокоманды. Применение макрокоманд полезно и даже необходимо, так как они позволяют получать компактный и структурированный код.

В заключение этого раздела обратитесь к файлам, прилагаемым к книге, где в каталоге данной главы вы найдете все исходные тексты, необходимые для сборки и запуска приложения (см. листинг 16.7). Собрав с помощью утилиты make.exe исполняемый файл, вы увидите, что реакция программы на выбор большинства

пунктов меню стандартная и заключается в выводе сообщения с использованием функции Win32 API MessageBox. Определена реакция лишь на два пункта меню Текст — пункты DrawText и TextOut. При их выборе на экран выводится текст. Тем самым демонстрируются различные способы вывода текста в окно приложения. Далее в материале этой и последующих глав мы определим реакцию и на остальные пункты меню.

Давайте теперь рассмотрим еще одну ключевую проблему программирования для Windows — *перерисовку изображения*. Для того чтобы понять ее смысл и важность, в любом порядке выберите пункты меню Текст ▸ DrawText и Текст ▸ TextOut На экране появятся строки текста. Далее сверните и разверните окно приложения. Строки текста исчезнут, оставив только строку, которая выводится как реакция на сообщение WM_PAINT. Причина этой ситуации в том, что Windows не хранит содержимое окна, и заботиться о его восстановлении после различных действий с окном должно приложение, создавшее это окно. Windows лишь посылает приложению сообщение WM_PAINT в случаях, когда с окном были произведены некоторые действия, например, свертывание-развертывание окна, изменение его размеров и т. д. Приложение, получив сообщение WM_PAINT, в качестве реакции на него должно обновить в необходимой степени содержимое своего окна. Более подробно эта проблема и пути ее решения рассмотрены в любом хорошем учебнике по программированию для Windows. Отметим лишь, что в конечном итоге все сводится к необходимости организовать вывод в окно приложения как реакцию на сообщение WM_PAINT. Существует несколько подходов к решению проблемы перерисовки. Мы рассмотрим наиболее общий и широко используемый.

Перерисовка изображения

Рассмотрим следующую абстракцию, цель которой — показать, что за содержимое окон на экране отвечают оконные процедуры тех приложений, которым эти окна принадлежат. Роль Windows в этом процессе минимальна и состоит в том, что при определенных действиях с окном оконной процедуре, отвечающей за связь с этим окном, посылается сообщение WM_PAINT. Получив это сообщение, оконная процедура должна уметь заново перерисовать содержимое всего окна или его части. Для общего случая это, если задуматься, не такая уж простая задача.

Если у вас возникают трудности с пониманием этого момента работы приложений в Windows, попробуйте развить абстракцию со школьной доской. Представьте себе школьный класс, в котором процесс обучения выглядит очень необычно. Вместо обычной школьной доски на стене есть некоторая *ограниченная область* (экран монитора). У каждого ученика (исполняемого файла) есть в портфеле своя доска (окно приложения), которая может (или не может, в зависимости от конструкции) менять свой размер. На всех учеников один комплект цветных мелков (контекст устройства), обладание которым разрешает ученику что-то рисовать на своей доске. Ученик объясняется с присутствующими графическими образами, выводя их на свою доску. Но прежде доску нужно повесить на стену в пределах *ограниченной области*. Для этого ученик, как ему и положено, подымает руку. Учитель (Windows) обязательно помогает ученику выйти к *ограниченной области* на стене (запускает на выполнение задачу) и повесить свою доску. Действия учителя и ученика в ходе этого процесса соответствуют определенному алгоритму и дей-

ствиям каркасного приложения. В результате могут быть закрыты частично или полностью доски учеников, вышедших к доске ранее и не ушедших пока на свои места в классе. При этом информация в скрытых частях расположенных ниже досок других учеников уничтожается в пределах *ограниченной области*.

Итак, доска ученика с помощью учителя повешена в пределах области на стене, и для того чтобы рисовать на ней, ученик должен попросить у учителя комплект мелков (см. функцию API GetDC или BeginPaint в листинге 16.4). Если ученик его получает, то он может начинать процесс рисования на своей доске. Как только ученик заканчивает этот процесс, он должен вернуть учителю комплект мелков (функция EndPaint). Далее ученик может закончить ответ и, забрав свою доску, занять свое место в классе (приложение завершило работу и «ушло» на диск). Если ученик делает это, то, забирая свою доску, он открывает доски других учеников, и... на этих досках обнаруживаются черные дыры, причем местоположение этих дыр соответствует местам, которые были скрыты доской их ушедшего товарища. Чтобы сгладить этот конфуз, учитель срочно оповещает об этом (Windows посылает сообщение WM_PAINT) учеников, у которых было испорчено содержимое досок. Каждый из этих учеников должен, попросив у учителя комплект мелков, перерисовать содержимое своей доски.

Данная абстракция довольно точно отражает логику работы Windows, причем не только в обозначенном нами контексте. Вы можете дополнить и развить ее в нужную вам сторону.

Как решается проблема перерисовки изображения практически? Для этого существует несколько способов, самый общий из которых заключается в использовании *виртуального окна*. Суть перерисовки изображения на основе виртуального окна заключается в использовании приложением некоторой области памяти для направления в него всего вывода программы. Реальный вывод в окно приложения осуществляется только как реакция на получение сообщения WM_PAINT. Не забывайте, что программа с помощью функции InvalidateRect() может сама себе послать сообщение WM_PAINT, когда ей потребуется вывести новый фрагмент изображения в окно приложения. Понятие виртуального окна настолько важно для организации работы Windows, что Win32 API содержит ряд функций, поддерживающих работу с этим окном: CreateCompatibleDC(), SelectObject(), GetStockObject(), BitBlt(), CreateCompatibleBitmap() и PatBlt(). Рассмотрим порядок их использования в реальном приложении.

Работа с виртуальным окном в программе организуется в два этапа: создание виртуального окна и организация непосредственной работы с ним. Создать виртуальное окно целесообразно при обработке сообщения WM_CREATE, то есть в момент создания окна приложения. Работать с этим окном можно в любое время, когда требуется вывод в окно.

Для наглядности обсуждения приведем фрагмент программы на языке C/C++:

```
//фрагмент оконной процедуры из программы на языке C/C++
...
case WM_CREATE:
//определить размеры экрана
maxX = GetSystemMetrics (SM_CXSCREEN);
maxY = GetSystemMetrics (SM_CXSCREEN);
//строим растровое изображение, совместимое с окном
```

```
hdc = GetDC (hwnd);
memdc = CreateCompatibleDC (hdc);
hbit = CreateCompatibleBitmap (hdc, maxX, maxY);
SelectObject (memdc, hbit);
//закрашиваем окно серым цветом
hbrush = GetStockObject (GRAY_BRUSH);
SelectObject (memdc, hbrush);
PatBlt (memdc, 0, 0, maxX, maxY, PATCOPY);
ReleaseDC (hwnd, hdc);
...
case WM_PAINT: //перерисовываем окно
hdc = BeginPaint (hwnd, &paintstruct); //получаем дескриптор реального окна
//копируем растровое изображение из памяти на экран
BitBlt (hdc, 0, 0, maxX, maxY, memdc, 0, 0, SRCCOPY);
EndPaint (hwnd, &paintstruct); //освобождаем дескриптор
...
//выводим в окно где-то в программе
TextOut(memdc, X, Y, str, strlen(str)); //выводим строку
//для немедленного вывода в окно вызываем функцию InvalidateRect:
InvalidateRect (hwnd, NULL, 1);
...
```

Физически виртуальное окно представляет собой растровое изображение, хранящееся в памяти. Работа с этой областью памяти организуется так же, как и с окном приложения на экране монитора. Это означает, что для работы с ним необходимо создать контекст устройства памяти, совместимый с контекстом окна. Это действие реализуется двумя функциями: с помощью функции GetDC() приложение получает контекст окна; функция CreateCompatibleDC() создает совместимый с контекстом окна контекст памяти memdc. После этого функцией CreateCompatibleBitmap() создается совместимое с реальным окном на экране растровое изображение. Его размеры должны соответствовать размеру окна, для работы с которым оно строится. Поэтому предварительно с помощью функции GetSystemMetrics() должны быть получены размеры окна и переданы как параметры в функцию CreateCompatibleBitmap(), которая возвращает дескриптор на созданное растровое изображение. После этого функция SelectObject() выбирает созданное растровое изображение в контекст памяти, который, в свою очередь, является совместимым с контекстом окна. Благодаря такой цепочке связей обращение к растровому изображению в памяти производится аналогично обращению к реальному окну. На практике это означает, что во всех функциях, выводящих изображение в окно, на месте параметра, соответствующего контексту устройства, необходимо указывать контекст устройства памяти. Например, функция TextOut() будет вызываться следующим образом:

```
push    lenTXT_TEXTOUT
push    offset @@TXT_TEXTOUT
push    150
push    10
push    @@memdc
call    TextOutA
```

В пятой строке этого фрагмента функции TextOutQ передается не контекст окна, а контекст виртуального окна, что и приводит к выводу не в реальное окно, а в виртуальное, являющееся растровым изображением. Как мы уже отметили, в программе есть только одно место, где производится вывод в реальное окно, — это фрагмент программы, обрабатывающий сообщение WM_PAINT. В случае виртуального

окна здесь располагается функция BitBlt(), которая копирует содержимое растрового изображения из контекста памяти в контекст реального окна. Таким образом постоянно обеспечивается актуальное содержимое окна приложения. В листинге 16.8 приведены фрагменты текста программы на языке ассемблера, демонстрирующей практическую реализацию способа перерисовки окна приложения с использованием виртуального окна. Полный текст программы находится среди файлов, прилагаемых к книге, в каталоге ..\prg16_3 данной главы.

Листинг 16.8. Фрагменты приложения prg16_3.asm

```
;Фрагменты приложения (prg16_3.asm) для Win32 с использованием меню
;и виртуального окна для перерисовки содержимого окна;
.386
locals;разрешает применение локальных меток (с префиксом @@) в программе
.model flat, STDCALL;модель памяти flat,
;STDCALL - передача параметров в стиле С (справа налево),
;вызываемая процедура чистит за собой стек
include windowA.inc ;включаемый файл с описаниями базовых структур
                    ;и констант Win32
include menu.inc    ;включаемый файл с определением имен пунктов
                    ;меню
;Объявление внешними используемых в данной программе
;функций Win32 (ASCII):
extrn   GetModuleHandleA:PROC
...
extrn   GetDC:PROC
extrn   BeginPaint :PROC
extrn   EndPaint:PROC
extrn   MessageBoxA:PROC
extrn   DrawTextA:PROC
extrn   GetClientRect:PROC
extrn   GetSystemMetrics:PROC
extrn   CreateCompatibleDC:PROC
extrn   CreateCompatibleBitmap:PROC
extrn   SelectObject:PROC
extrn   GetStockObject:PROC
extrn   PatBlt:PROC
extrn   BitBlt:PROC
extrn   InvalidateRect:PROC
extrn   DeleteDC:PROC
...
.data
memdc   dd  0   ;!!!это глобальная переменная
maxX    dd  0   ;!!!это глобальная переменная
maxY    dd  0   ;!!!это глобальная переменная
...
lpRect  RECT<?>
...
.code
start   proc near
;точка входа в программу:
...
WinMain:
...
start   endp
;------------------WindowProc------------------------------------------
WindowProc  proc
arg @@hwnd:DWORD, @@mes:DWORD, @@wparam:DWORD, @@lparam:DWORD
uses ebx, edi, esi, ebx ;эти регистры обязательно должны сохраняться
local   @@hdc:DWORD, @@hbrush:DWORD, @@hbit:DWORD
        cmp @@mes, WM_DESTROY
```

```
    je   wmdestroy
    cmp @@mes, WM_CREATE
    je   wmcreate
    cmp @@mes, WM_PAINT
    je   wmpaint
    crap @@mes, WM_COMMAND
    je   wmcommand
    jmp default
wmcreate:
;создание растрового изображения, совместимого с окном приложения
;получим размер экрана в пикселах int GetSystemMetrics(int nIndex)
    push    SM_CXSCREEN
    call    GetSystemMetrics
    mov     maxX, eax
    push    SM_CYSCREEN
    call    GetSystemMetrics
    mov     maxY, eax
;лолучить контекст устройства окна на экране @@hdc=GetDC(@@hwnd)
    push    @@hwnd
    call    GetDC
    mov     @@hdc, eax
;получить совместимый контекст устройства памяти
;memdc=CreateCompatibleDC(@@hdc)
    push    @@hdc
    call    CreateCompatibleDC
    mov     memdc, eax  ;!!! memdc - глобальная переменная
;получить дескриптор растрового изображения в памяти
; @@hbit=CreateCompatibleBitmap(@@hdc, @@maxX, @@maxY)
    push    maxY
    push    maxX
    push    @@hdc
    call    CreateCompatibleBitmap
    mov     @@hbit, eax
;выбираем растр в контекст памяти SelectObject(memdc, @@hbit)
    push    @@hbit
    push    memdc
    call    SelectObject
;выполним первичное заполнение растра серым цветом
;получим дескриптор серой кисти hbrush=GetStockObject(GRAY_BRUSH)
    push    GRAY_BRUSH
    call    GetStockObject
    mov     @@hbrush, eax
;выбираем кисть в контекст памяти SelectObject(memdc, @@hbrush)
    push    @@hbrush
    push    memdc
    call    SelectObject
;заполняем выбранной кистью виртуальное окно
;BOOL PatBlt(HDC hdc, int nXLeft. int nYLeft, int nWidth,
    ;int nHeight, DWORD dwRop)
    push    PATCOPY
    push    maxY
    push    maxX
    push    NULL
    push    NULL
    push    memdc
    call    PatBlt
;освободим контекст устройства ReleaseDC(@@hwnd, @@hdc)
    push    @@hdc
    push    @@hwnd
    call    ReleaseDC
;обозначим создание окна звуковым эффектом
;готовим вызов функции BOOL PlaySound(LPCSTR pszSound,
    ;HMODULE hmod, DWORD fdwSound)
    push    SND_SYNC+SND_FILENAME
```

продолжение ⇨

Листинг 16.8 *(продолжение)*

```
        push    NULL
        push    offset playFileCreate
        call    PlaySoundA
;возвращаем значение 0
        mov     eax, 0
        jmp     exit_wndproc
wmpaint:
;получим контекст устройства HDC BeginPaint(HWND hwnd,
        ;LPPAINTSTRUCT lpPaint);
        push    offset  ps
        push    @@hwnd
        call    BeginPaint
        mov     @@hdc, eax
;обозначим перерисовку окна звуковым эффектом
        push    SND_SYNC+SND_FILENAME
        push    NULL
        push    offset playFilePaint
        call    PlaySoundA
;выведем строку текста в окно BOOLTextOut(HDChdc,intnXStart,
;int nYStart, LPCTSTRlpString,intcbString);
        push    MesWindowLen
        push    offset MesWindow
        push    100
        push    10
        push    memdc
        call    TextOutA
;вывод виртуального окна в реальное окно
;BOOL BitBlt(HDC hdcDest, int nXDest, int nYDest, int nWidth,
;int nHeight, HDC hdcSrc, int nXSrc, int nYSrc, DWORD dwRop)
        push    SRCCOPY
        push    NULL
        push    NULL
        push    memdc
        push    maxY
        push    maxX
        push    NULL
        push    NULL
        push    @@hdc
        call    BitBlt
;освободить контекст BOOL EndPaint(HWND hWnd,
        ;CONST PAINTSTRUCT *lpPaint);
        push    offset ps
        push    @@hwnd
        call    EndPaint
        mov     eax, 0  ;возвращаемое значение - 0
        jmp     exit_wndproc
wmdestroy:
...
wmcommand:
;вызов процедуры обработки сообщений от меню
;MenuProc (DWORD @@hwnd, DWORD @@wparam)
        push    @@wparam
        push    @@hwnd
        call    MenuProc
        jmp     exit_wndproc
default:
;...
exit_wndproc:
        ret
WindowProc  endp
;-----------------MenuProc---------------------------------------
;обработка сообщений от меню
```

```
MenuProcproc
arg @@hwnd:DWORD, @@wparam:DWORD
uses ebx
local
    mov ebx, @@wparam   ;в Ьх идентификатор меню
...
    jmp @@exit
@@idmdrawtext:
;получим размер рабочей области BOOL GetClientRect(HWND hWnd,
    ;LPRECT lpRect);
    push    offset IpRect
    push    @@hwnd
    call    GetClientRect
;выведем строку текста в окно int DrawText(HDC hDC,
    ;LPCTSTR lpString, int nCount, length,
;LPRECT IpRect, UINT uFormat);
    push    DT_SINGLELINE+DT_BOTTOM
    push    offset IpRect
    push    -1
    push    offset @@TXT_DRAWTEXT
    push    memdc
    call    DrawTextA
;генерация сообщения WM_PAINT для вывода строки на экран
;BOOL InvalidateRect(HWND hWnd, CONST RECT *lpRect, BOOL bErase)
    push    1
    push    NULL
    push    @@hwnd
    call    InvalidateRect
    jmp     @@exit
@@idmtextout:
;выведем строку текста в окно BOOL TextOut(HDC hdc, int nXStart,
;int nYStart, LPCTSTR IpString, int cbString)
    push    lenTXT_TEXTOUT
    push    offset @@TXT_TEXTOUT
    push    150
    push    10
    push    memdc
    call    TextOutA
;генерация сообщения WM_PAINT для вывода строки на экран
    push    0
    push    NULL
    push    @@hwnd
    call    InvalidateRect
    jmp     @@exit
@@idmlength:
...
    jmp     @@exit
@@idmrectangle:
...
    jmp     @@exit
@@idmpeacock:
...
    jmp     @@exit
@@idmlaces:
...
    jmp     @@exit
@@idmabout:
...
    jmp     @@exit
;...
@@exit:
    mov    eax, 0
    ret
@@TXT_ABOUT db  'IDM_ABOUT', 0
```

продолжение ⇨

Листинг 16.8 *(продолжение)*

```
@@TXT_LACES db 'IDM_LACES', 0
 @@TXT_PEACOCK  db  'IDM_PEACOCK', Э
@@TXT_RECTANGLE db  'IDM_RECTANGLE', 0
@@TXT_LENGTHdb  'IDM_LENGTH', 0
@@TXT_TEXTOUT   db  'Текст выведен функцией TEXTOUT'
lenTXT_TEXTOUT=$-@@TXT_TEXTOUT
@@TXT_DRAWTEXT  db  'Текст выведен функцией DRAWTEXT', 0
MenuProcendp
end start
```

Возможно, результаты работы программы из листинга 16.8 вам покажутся не очень красивыми, но такая цель и не ставилась. Назначение этой программы — исследовательское. Из-за задержек, вызванных воспроизведением звуковых файлов, хорошо виден момент перерисовки окна. Такую технологию можно использовать для более глубокого исследования механизмов работы Windows.

Окна диалога в Windows-приложениях

Окна диалога являются важными и популярными элементами пользовательского интерфейса Windows. Редкое оконное приложение обходится без окон этого типа. Физически окно диалога представляет собой специфическое окно, работа с которым поддерживается на уровне интерфейса Win32 API Windows. Основное назначение этого окна — помочь пользователю сформировать информацию, необходимую для управления работой приложения. Наиболее наглядные примеры окон этого типа — окна диалога в текстовом редакторе. С их помощью можно задать параметры шрифта, страницы или печати. Очень важно, что разработка таких окон не требует программирования. Для описания окон диалога система Windows поддерживает специальный тип ресурса. Более подробно о том, как в программе должны выглядеть окна диалога, из каких элементов они состоят, а также о деталях управления этими окнами вы можете почитать в других источниках. Наша цель — показать, каким образом работа с окнами диалога организуется программой, написанной на языке ассемблера.

С точки зрения технологии, организация работы с окнами диалога, реализуемая программой на ассемблере, ничем не отличается от того, как это делается на любом другом языке. Чтобы создать окно диалога на языке ассемблера, необходимо выполнить следующие шаги.

1. Описать окно диалога в файле ресурсов.
2. Разработать *диалоговую* функцию, которая будет обрабатывать сообщения, предназначенные для определенного в файле ресурсов окна диалога.
3. Активизировать окно диалога в приложении.

Для того чтобы разговор был предметным, поставим себе цель разработать конкретную программу. В последней программе (см. листинг 16.8) мы предусмотрели возможность ее расширения. Сейчас для этого настало время. Дополним программу из листинга 16.8 фрагментами, обеспечивающими работу подменю Примитивы в меню Графика (см. рис. 16.2). Это подменю обеспечивает доступ к пунктам меню Отрезок и Прямоугольник. Пусть они инициируют открытие окон диалога, предназначенных для настройки параметров простейших фигур (отрезка и прямоуголь-

ника) и прорисовки их в окне приложения в соответствии с заданными параметрами. Параллельно мы разберемся с некоторыми общими принципами работы с графикой, понимая которые, вы сможете реализовать более сложные алгоритмы.

Описание окна диалога в файле ресурсов

Как было отмечено, окно диалога представляет собой специальный объект, призванный облегчить взаимодействие пользователя с выполняющейся программой и настроить ее на определенные условия функционирования. Окно диалога состоит из элементарных объектов, называемых *элементами управления*. Система Windows поддерживает несколько типов таких объектов. В рассматриваемом далее примере мы будем использовать только небольшую их часть.

Для описания внешнего вида окна диалога и его интерфейса с приложением используется специальный тип ресурса — DIALOG. В отличие от ресурса меню, который можно создать вручную, ресурс окна диалога лучше создавать с помощью соответствующих программных средств — редакторов ресурсов. Основная причина здесь в том, что при ручном определении размеров и взаимного расположения элементов управления трудно представить, как окно будет выглядеть на экране. Сформулируем некоторые общие принципы, которые позволят вам без труда выполнять эту часть работы.

Пакет ассемблера не имеет своего редактора ресурсов, поэтому вам придется поискать его на стороне. Это не обязательно должен быть автономный редактор ресурсов — вполне подойдет редактор, встроенный в интегрированную среду разработки какого-нибудь языка высокого уровня. Главное, чтобы он создавал файл описания ресурса с расширением .rc. В файле описания наряду со строками, описывающими ресурс, будут строки, необходимые для работы интегрированной среды с ресурсами. Их нужно просто удалить. Строки, содержащие непосредственное описание ресурса, включите в общий файл описания ресурсов приложения.

Ресурсы окон диалога показанного далее приложения создавались с помощью редактора ресурсов VC ++ версии 6.0. Ресурс каждого окна диалога готовился отдельно, а затем добавлялся в файл ресурсов приложения prg16.4.rc. В этом же файле, кстати, уже находился ранее разработанный ресурс с описанием меню приложения. Параллельно с созданием файла ресурсов приложения необходимо создать (или модифицировать) включаемый файл описаний, содержащий символьные константы для пунктов меню и окон диалога. В нашем примере файл ресурсов имеет описание, представленное в листинге 16.9.

Листинг 16.9. Файл ресурсов prg16_4.rc

```
#include   "Prg16_4.h"
#include   <windows.h>
IDI_ICON1  ICONDISCARDABLE "icon1.ico"
MYMENU MENU DISCARDABLE
{
 см. описание меню в пункте "Меню в Windows-приложениях"
 }
 ///////////////////////////////////////////////////////////////////////
 II
 // Dialog для отрезка
 //
  IDD_DIALOG1 DIALOG DISCARDABLE 0, 0, 186, 95
```

продолжение.

Листинг **16.9** ***(продолжение)***

```
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Отрезок"
FONT 8, "MS Sans Serif"
BEGIN
    DEFPUSHBUTTON  "OK", IDOK, 35, 72, 50, 14
    PUSHBUTTON  "Cancel", IDCANCEL, 118, 72, 50, 14
    LTEXT   "Задайте координаты концов отрезка:", IDC_STATIC, 22,
        6, 134, 8
    EDITTEXTIDC_EDIT1, 34, 37, 20, 12, ES_AUTOHSCROLL
    LTEXT   "Xstart", IDC_STATIC, 5, 40, 19, 8
    LTEXT   "Ystart", IDC_STATIC, 5, 54, 19, 8
    LTEXT   "Xend", IDC_STATIC, 91, 39, 18, 8
    LTEXT   "Yend", IDC_STATIC, 91, 52, 18, 8
    EDITTEXTIDC_EDIT2, 34, 52, 20, 12, ES_AUTOHSCROLL
    EDITTEXTIDC_EDIT3, 118, 36, 20, 12, ES_AUTOHSCROLL
    EDITTEXTIDC_EDIT4, 118, 52, 20, 12, ES_AUTOHSCROLL
END
/////////////////////////////////////////////////////////////////////////
//
// Dialog для прямоугольника
//
IDD_DIALOG2 DIALOG DISCARDABLE  0, 0, 186, 95
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Прямоугольник"
FONT 8, "MS Sans Serif"
BEGIN
    DEFPUSHBUTTON  "OK", IDOK, 35, 72, 50, 14
    PUSHBUTTON  "Cancel". IDCANCEL, 118, 72, 50, 14
    LTEXT   "Задайте координаты углов прямоугольника:",
        IDC_STATIC, 22, 6, 174, 8
    EDITTEXTIDC_EDIT1, 34, 37, 20, 12, ES_AUTOHSCROLL
    LTEXT   "X", IDC_STATIC, 22, 39, 8, 8
    LTEXT   "Y", IDC_STATIC, 21, 55, 8, 8
    LTEXT   "X", IDC_STATIC, 104, 39, 8, 8
    LTEXT   "Y", IDC_STATIC, 104, 54, 8, 8
    EDITTEXTIDC_EDIT2, 34, 52, 20, 12, ES_AUTOHSCROLL
    EDITTEXTIDC_EDIT3, 118, 36, 20, 12, ES_AUTOHSCROLL
    EDITTEXTIDC_EDIT4, 118, 52, 20, 12, ES_AUTOHSCROLL
    LTEXT   "Left Top:", IDC_STATIC, 4, 27, 32, 8
    LTEXT   "Right Bottom:", IDC_STATIC, 65, 27, 46, 8
END
/////////////////////////////////////////////////////////////////////////
//
// Dialog для пункта меню About
//
AboutBox DIALOG 20, 20. 160, 80
  STYLE WS_POPUP | WS_DLGFRAME
  {
    CTEXT   "TASM32"-1, 0. 12, 160, 8
    ICON"IDI_ICON1" -1, 8, 8, 0, 0
    CTEXT   "Win32 Demo Program"-1, 0, 36, 160, 8
    CTEXT   "(c) Юров Виктор, 2003" -1, 0, 48, 160, 8
    DEFPUSHBUTTON  "OK"IDOK, 64, 60, 32, 14, WS_GROUP
  }
```

В файле определены ресурсы трех типов: значок (ICON), меню (MENU) и окно диалога (DIALOG). Их можно создать по отдельности в редакторе ресурсов, а затем с помощью текстового редактора объединить в один файл. При этом вы не должны забывать и о сопутствующим их включаемым файлам с расширением .h. Их также удобно объединить в единый файл так, как это сделано для нашего примера (листинг 16.10).

Листинг 16.10. Включаемый файл с идентификаторами элементов ресурсов prg16_4.h

```
#define IDM_DRAWTEXT100
#define IDM_TEXTOUT      101
#define IDM_LENGTH       102
#define IDM_RECTANGLE    103
#define IDM_PEACOCK      104
#define IDM_LACES        105
#define IDM_ABOUT        106

#define IDC_EDIT1         1000
#define IDC_EDIT2         1001
#define IDC_EDIT3         1002
#define IDC_EDIT4         1003
#define IDC_STATIC       -1
```

На основании включаемого файла должен быть составлен эквивалентный включаемый файл prg16_4.inc (листинг 16.11). Директиву include prg16_4.inc необходимо помещать в начале исходного текста приложения. В принципе, этого можно и не делать, но тогда нельзя будет использовать символические имена констант, определенные в файле prg16_4.h. Вместо них в соответствующих местах программы придется указывать их численные значения.

Листинг 16.11. Включаемый файл с идентификаторами элементов ресурсов prg16_4.inc

```
IDM_DRAWTEXT    equ 100
IDM_TEXTOUT     equ 101
IDM_LENGTH      equ 102
IDM_RECTANGLE   equ 103
IDM_PEACOCK     equ 104
IDM_LACES       equ 105
IDM_ABOUT       equ 106
IDC_EDIT1       equ 1000
IDC_EDIT2       equ 1001
IDC_EDIT3       equ 1002
IDC_EDIT4       equ 1003
IDC_STATIC      equ -1
```

После того как файлы созданы, необходимо выполнить компиляцию файла ресурсов и получить его двоичный эквивалент prg16_4.res. Файл ресурсов рассматриваемого нами приложения, в отличие от файла ресурсов prg16_3.rc, имеет особенности. Эти особенности связаны с тем, что при описании ресурса окна диалога используются символические имена констант, определенные в файле windows.h. Дальнейшие действия зависят от того, каким компилятором языков С/С++ вы располагаете, хотя последовательность этих действий во всех случаях будет приблизительно одинаковой. Рассмотрим эту последовательность на примере компилятора Visual C++ версии 6.0, который использовался для компиляции файла ресурсов приложения prg16_4:

1. Скопировать исполняемый файл компилятора ресурсов ..\Msdev\bin\rc.exe в свой рабочий каталог ..\..\WORK32[1].

[1] Предполагается, что вы следуете рекомендациям и ведете всю текущую работу в рабочем каталоге, к примеру ..\..\WORK. В нем находятся необходимые файлы из пакета TASM и все файлы, относящиеся к текущему разрабатываемому приложению. Теперь у вас будет два каталога: для 16-разрядных приложений ..\..\WORK и для 32-разрядных приложений ..\..\WORK32.

2. Поместить в файл autoexec.bat строку: callTasmVars.bat. Файл TasmVars.bat предназначен для установки переменных окружения. Значения переменных определяют путь для поиска включаемых, исполняемых и других файлов. После модификации файла autoexec.bat необходимо перезагрузить компьютер, с тем чтобы изменения вступили в силу.
3. Запустить компилятор ресурсов, указав ему в качестве параметра имя созданного нами ранее ресурсного файла:

```
rc.exe prg16_4.rc
```

Если указанные действия выполнены корректно, вы получите файл prg16_4.res. То, насколько удачно вам удалось определить ресурсы для вашего приложения, можно проверить только на этапе его выполнения. Если что-то вас не устраивает, то описанный ранее процесс придется повторить, внеся необходимые коррективы.

Диалоговая процедура

В процессе работы с окном диалога пользователь выполняет некоторые действия, о которых с помощью сообщений становится известно приложению. В приложении для каждого окна диалога должна существовать своя процедура, предназначенная для обработки сообщений этого окна. Эта процедура называется *диалоговой процедурой*. Даже самое примитивное окно диалога содержит элемент, сообщение от которого поступает в диалоговую процедуру. Обычно это кнопка OK или Cancel. На языках C/C++ соответствующая диалоговая процедура выглядит так, как показано в листинге 16.12.

Листинг 16.12. Диалоговая процедура на языках C/C++

```
BOOL CALLBACK DialogProc (HWND hdwnd, UINT message, WPARAM wParam,
    LPARAM lParam)
{
    switch(message)
    {
        case WM_INITDIALOG:
            return 1;
        case WM_COMMAND:
        switch (LOWORD (wParam))
        {
        case IDOK:  //или IDCANCEL
            EndDialog(hdwnd, 0);
            return 1;
        }
    }
    return 0;
}
```

В приложении prg16.4.asm, частично представленном в листинге 16.13, имеется фрагмент (строки 523-549), являющийся эквивалентом программы из листинга 16.12 на языке ассемблера. Это процедура AboutDialog, поддерживающая работу окна диалога AboutBox, описанного в файле ресурсов (см. листинг 16.9). Заметьте, что если оконная процедура самостоятельно обрабатывает сообщение, то она должна возвратить 1 (return 1), если нет, то 0 (return 0).

Листинг 16.13. Фрагменты приложения prg16_4.asm

```
<1> ;prg16_4.asm
<2> ;Пример приложения для Win32 с использованием меню,
<3> ;окон диалогов, решением проблемы перерисовки содержимого окна
<4> ;и демонстрацией некоторых принципов работы с графикой
;Внимание!!! Координаты вводить из четырех знаков с ведущими нулями:
;например, для ввода числа 12 вводить 0012, для 123 вводить 0123 и т. д.
<5> .486
<6> locals;разрешает применение локальных меток (с префиксом @@) в программе
<7> .model flat, STDCALL ;модель памяти flat,
<8> ;STDCALL - передача параметров в стиле С (справа налево),
<9> ;вызываемая процедура чистит за собой стек
<10>    include windowA.inc ;включаемый файл с описаниями базовых структур
                            ;и констант Win32
<11>    include prg16_4.inc ;включаемый файл с определением имен
                            ;пунктов меню и окна диалога
<12>    ;Объявление внешними используемых в данной программе
        ;функций Win32 (ASCII):
<13>    extrn   GetModuleHandleA:PROC
<...> см. исходный текст программы
<55>    extrn   GetDlgItemTextA:PROC
<56>    ;объявление оконной и диалоговых функций объектами,
        ;видимыми за пределами данного кода
<57>    public  WindowProc
<58>    public  DialogProc1
<59>    public  DialogProc2
<60>    .data
<61>    Xstart  dd  0, 0
<62>    Xend    dd  0, o
<63>    Ystart  dd  0, 0
<64>    Yend    dd  o, o
<65>    hwnd    dd  0
<66>    hInst   dd  0
<67>    memdc   dd  o   ;!!!это глобальная переменная
<68>    maxX    dd  0   ;!!!это глобальная переменная
<69>    maxY    dd  0   ;!!!это глобальная переменная
<70>    ;lpVersionInformation   OSVERSIONINFO   <?>
<71>    wcl WNDCLASSEX  <?>
<72>    message MSG <?>
<73>    ps  PAINTSTRUCT <?>
<74>    lpRect  RECT<?>
<75>    pt  POINT   <?>
<76>    szClassName db  'Приложение Win32', 0
<77>    szTitleName db  'Каркасное приложение Win32 на ассемблере', 0
<78>    MesWindow   db  'Привет! Ну как вам процесс разработки приложения
                         на ассемблере?'
<79>    MesWindowLen=   $-MesWindow
<80>    ;звуковые файлы
<81>    playFileCreate  db  'create.wav', 0
<82>    playFilePaint   db  'paint.wav', 0
<83>    playFileDestroy db  'destroy.wav', 0
<84>    ;имена ресурсов:
<85>    lpmenu  db  "MYMENU", 0
<86>    lpdlg1  db  "IDD_DIALOG1", 0
<87>    lpdlg2  db  "IDD_DIALOG2", 0
<88>    lpdlg3  db  "AboutBox", 0
<89>    ;переменные для макроса show_eax
<90>    eedx        dd  0
<91>    eecx        dd  0, 0
<92>    template    db  '0123456789ABCDEF'
<93>    MesMsgBox   db  'Отладка (содержимое eax): ' , 0
<94>    ;описание макрокоманд
<95>    include show_eax.inc
```

продолжение ⇨

Листинг **16.13** *(продолжение)*

```
sim4_to_EAXbin  macro   sim4:req
    local   ml
    push    eax
    push    ebx
    push    ecx
    mov     ebx, 1
    mov     eax, sim4
    bswap   eax
    mov     sim4, 0
    push    eax
    mov     ecx, 4
ml: and     eax, 0fh
    imul    eax, ebx
    imul    ebx, 10
    add     sim4, eax
    pop     eax
    shr     eax, 8
    push    eax
    loop    ml
    pop     eax
    pop     ecx
    pop     ebx
    pop     eax
    endm
.code
start   proc near
;точка входа в программу:
;начало стартового кода
...
;конец стартового кода
WinMain:
;определить класс окна ATOM RegisterClassEx(CONST WNDCLASSEX
;*lpWndClassEx), где *lpWndClassEx - адрес структуры WndClassEx
;...
    call    RegisterClassExA
    test    ax, ax      ;проверить на успех регистрации класса окна
    jz      end_cycl_msg;неудача
;создаем окно:
;...
    call    CreateWindowExA
    mov     hwnd, eax   ;hwnd - дескриптор окна
;показать окно:
;...
    call    ShowWindow
;перерисовываем содержимое окна
;...
    call    UpdateWindow
;запускаем цикл сообщений:
;...
cycljnsg:
;...
    call    GetMessageA
    cmp     ax, 0
    je      end_cycl_msg
;трансляция ввода с клавиатуры
;...
    call    TranslateMessage
;отправляем сообщение оконной процедуре
;...
    call    DispatchMessageA
    jmp     cycl_msg
end_cycl_msg:
```

```
;выход из приложения
;...
    call    ExitProcess
start   endp

; --..........-....WindowProc----------------------------.-.....
WindowProc  proc
arg @@hwnd:DWORD, @@mes:DWORD, @@wparam:DWORD, @@lparam:DWORD
uses ebx, edi, esi, ebx ;эти регистры обязательно должны сохраняться
local   @@hdc:DWORD, @@hbrush:DWORD, @@hbit:DWORD
    cmp @@mes, WM_DESTROY
    je  wmdestroy
    cmp @@mes, WM_CREATE
    je  wmcreate
    cmp @@mes, WM_PAINT
    je  wmpaint
    cmp @@mes, WM_COMMAND
    je  wmcommand
    jmp default
wmcreate:
;создание растрового изображения, совместимого с окном приложения
;...
;обозначаем создание окна звуковым эффектом
;...
    call    PlaySoundA
;возвращаем значение 0
    mov eax, 0
    jmp exit_wndproc
wmpaint:
;...
;обозначаем перерисовку окна звуковым эффектом
;...
;вывод виртуального окна в реальное окно
;...
wmdestroy:
;удалить виртуальное окно DeleteDC(memdc)
;...
;послать сообщение WM_QUIT
;...
wmcommand:
;вызов процедуры обработки сообщений от меню
;MenuProc (DWORD @@hwnd, DWORD @@wparam)
    push    @@wparam
    push    @@hwnd
    call    MenuProc
    jmp     exit_wndproc
default:
;обработка по умолчанию
;...
    jmp     exit_wndproc
;...
exit_wndproc:
    ret
WindowProc  endp
;..........-.......MenuProc................-----...............
;обработка сообщений от меню
MenuProcproc
arg @@hwnd:DWORD, @@wparam:DWORD
useseax, ebx
    mov ebx, @@wparam   ;в Ьх идентификатор меню
    cmp bx, IDM_DRAWTEXT
    je  @@idmdrawtext
    cmp bx, IDM_TEXTOUT
    je  @@idmtextout
```

продолжение ↗

Листинг **16.13** *(продолжение)*

```
        cmp bx, IDM_LENGTH
        je  @@idmlength
        cmp bx, IDM_RECTANGLE
        je  @@idmrectangle
        cmp bx, IDM_PEACOCK
        je  @@idmpeacock
        cmp bx, IDM_LACES
        je  @@idmlaces
        cmp bx, IDM_ABOUT
        je  @@idmabout
        jmp @@exit
    @@idmdrawtext:
    ;получим размер рабочей области BOOL GetClientRect (HWND hWnd,
    ;LPRECT lpRect);
        push    offset IpRect
        push    @@hwnd
        call    GetClientRect
    ;выводим строку текста в окно int DrawText(HDC hDC,
    ;LPCTSTR lpString, int nCount,
    ; LPRECT IpRect, UINT uFormat);
        push    DT_SINGLELINE+DT_BOTTOM
        push    offset IpRect
        push    -1
        push    offset @@TXT_DRAWTEXT
        push    memdc
        call    DrawTextA
    ;генерация сообщения WM_PAINT для вывода строки на экран
        push    1
        push    NULL
        push    @@hwnd
        call    InvalidateRect
        jmp     @@exit
    @@idmtextout:
    ;выводим строку текста в окно BOOL TextOut(HDC hdc, int nXStart,
    ;int nYStart, LPCTSTR IpString, int cbString);
    ;...
        call    TextOutA
    ;генерация сообщения WM_PAINT для вывода строки на экран
        push    0
        push    NULL
        push    @@hwnd
        call    InvalidateRect
        jmp     @@exit
    @@idmlength:
    ;вызываем окно диалога int DialogBoxParam(HINSTANCE hInstance,
    ;LPCTSTR lpTemplateName,
    ;HWND hWndParent, DLGPROC lpDialogFunc, LPARAM dwInitParam)
        push    0
        push    offset DialogProcl
        push    @@hwnd
        push    offset lpdlg1
        push    hInst
        call    DialogBoxParamA
    ;установить текущую точку BOOL MoveToEx(HDC hdc, int X, int Y,
    ;LPPOINT lpPoint)
        push    NULL
        push    Ystart
        push    Xstart
        push    memdc
        call    MoveToEx
    ;вывод линии BOOL LineTo(HDC hdc, int nXEnd, int nYEnd)
        push    Yend
```

```
       push    Xend
       push    memdc
       call    LineTo
   ;генерация сообщения WM_PAINT для вывода строки на экран
       push    0
       push    NULL
       push    @@hwnd
       call    InvalidateRect
       jmp     @@exit
   @@idmrectangle:
   ;вызываем окно диалога
       push    0
       push    offset DialogProc2
       push    @@hwnd
       push    offset lpdlg2
       push    hInst
       call    DialogBoxParamA
   ;вывод прямоугольника BOOL Rectangle(HDC hdc, int nLeftRect,
   ;int nTopRect, int nRightRect, int nBottomRect)
       push    Ystart
       pop     eax
       push    eax
       show_eax
       push    Xstart
       pop     eax
       push    eax
       show_eax
       push    Yend
       pop     eax
       push    eax
       show_eax
       push    Xend
       pop     eax
       push    eax
       show_eax
       push    memdc
       call    Rectangle
   ;генерация сообщения WM_PAINT для вывода строки на экран
       push    0
       push    NULL
       push    @@hwnd
       call    InvalidateRect
       jmp     @@exit
   @@idmpeacock:
       push    MB_ICONINFORMATION+MB_OK
       push    offset szTitleName
       push    offset @@TXT_PEACOCK
       push    @@hwnd
       call    MessageBoxA
       jmp     @@exit
   @@idmlaces:
       push    MB_ICONINFORMATION+MB_OK
       push    offset szTitleName
       push    offset @@TXT_LACES
       push    @@hwnd
       call    MessageBoxA
       jmp     @@exit
   @@idmabout:
   ;вызываем окно диалога
       push    0
       push    offset AboutDialog
       push    @@hwnd
       push    offset lpdlg3
       push    hInst
```

продолжение ⇒

Листинг 16.13 *(продолжение)*

```
       call     DialogBoxParamA
       jmp      @@exit
;...
@@exit:
     mov eax, 0
     ret
@@TXT_ABOUT     db  'IDM_ABOUT', 0
@@TXT_LACES db  'IDM_LACES', 0
@@TXT_PEACOCK   db  'IDM_PEACOCK', 0
@@TXT_TEXTOUT   db  'Текст выведен функцией TEXTOUT'
lenTXT_TEXTOUT=$-@@TXT_TEXTOUT
@@TXT_DRAWTEXT  db  'Текст выведен функцией DRAWTEXT', 0
MenuProcendp
;-----------------DialogProc1---------------------------------------
DialogProcl proc
arg @@hdlg:DWORD, @@message:DWORD, @@wparam:DWORD, @@lparam:DWORD
uses eax, ebx, edi, esi
     mov eax, @@message
     cmp ax, WM_INITDIALOG
     je  @@wminitdialog
     cmp ax, WM_COMMAND
      jne @@exit_false
    mov ebx, @@wparam;в bx идентификатор элемента управления
     cmp bx, IDOK
     je  @@idok
     cmp bx, IDCANCEL
     je  @@idcancel
     jmp @@exit_false
@@wminitdialog:
     jmp @@exit_true
@@idok:
;читаем Xstart UINT GetDlgItemText(HWND hDlg, int nIDDlgItem,
;LPTSTR lpString, int nMaxCount);
     push    5
     push    offset Xstart
     push    IDC_EDIT1
     push    @@hdlg
     call    GetDlgItemTextA
     push    MB_ICONINFORMATION+MB_OK
<...>
     sim4_to_EAXbin  Xstart
;читаем Ystart
     push5
     push    offset Ystart
     push    IDC_EDIT2
     push    @@hdlg
     call    GetDlgltemTextA
<...>
     sim4_to_EAXbin  Ystart
;читаем Xend
     push    5
     push    offset Xend
     push    IDC_EDIT3
     push    @@hdlg
     call    GetDlgltemTextA
<...>
     sim4_to_EAXbin  Xend
;читаем Yend
     push    5
     push    offset Yend
     push    IDC_EDIT4
     push    @@hdlg
```

```
    call    GetDlgItemTextA
<...>
    sim4_to_EAXbin  Yend
    push    0
    push    @@hdlg
    call    EndDialog
    jmp     @@exit_true
@@idcancel:
    push    NULL
    push    @@hdlg
    call    EndDialog
    jmp     @@exit_true
@@exit_false:
    mov eax, 0
    ret
@@exit_true:
    mov eax, 1
    ret
DialogProc1 endp
;-------------------DialogProc2---------------------------------
DialogProc2 proc
arg @@hdlg:DWORD, @@message:DWORD, @@wparam:DWORD, @@lparam:DWORD
uses eax, ebx, edi, esi
    mov eax, @@message
    cmp ax, WM_INITDIALOG
    je  @@wminitdialog
    cmp ax, WM_COMMAND
    jne @@exit_false
    mov ebx, @@wparam;в bx идентификатор элемента управления
    cmp bx, IDOK
    je  @@idok
    cmp bx, IDCANCEL
    je  @@idcancel
    jmp @@exit_false
@@wminitdialog:
    jmp @@exit_true
@@idok:
;читаем Xstart UINT GetDlgItemText(HWND hDlg, int nIDDlgItem,
;LPTSTR lpString, int nMaxCount);
    push    5
    push    offset Xstart
    push    IDC_EDIT1
    push    @@hdlg
    call    GetDlgItemTextA
<...>
    sim4_to_EAXbin  Xstart
;читаем Ystart
    push    5
    push    offset Ystart
    push    IDC_EDIT2
    push    @@hdlg
    call    GetDlgItemTextA
<...>
    sim4_to_EAXbin  Ystart
;читаем Xend
    push    5
    push    offset Xend
    push    IDC_EDIT3
    push    @@hdlg
    call    GetDlgItemTextA
<...>
    sim4_to_EAXbin  Xend
;читаем Yend
    push    5
```

продолжение ⇒

Листинг 16.13 *(продолжение)*

```
    push    offset Yend
    push    IDC_EDIT4
    push    @@hdlg
    call    GetDlgItemTextA
<...>
    sim4_to_EAXbin  Yend
    push    NULL
    push    @@hdlg
    call    EndDialog
    jmp     @@exit_true
@@idcancel:
    push    NULL
    push    @@hdlg
    call    EndDialog
    jmp     @@exit_true
@@exit_false:
    mov     eax, 0
    ret
@@exit_true:
    mov eax, 1
    ret
DialogProc2 endp
;--------------------AboutDialog--------------------------------------
AboutDialog proc
arg @@hdlg:DWORD, @@message:DWORD, @@wparam:DWORD, @@lparam:DWORD
uses eax, ebx, edi, esi
    mov eax, @@message
    cmp ax, WM_INITDIALOG
    je  @@wminitdialog
    cmp ax, WM_COMMAND
    jne @@exit_false
    mov ebx, @@wparam;в bx идентификатор элемента управления
    cmp bx, IDOK
    je  @@idok
    jmp @@exit_false
@@wminitdialog:
    jmp @@exit_true
@@idok:
    push    NULL
    push    @@hdlg
    call    EndDialog
    jmp     @@exit_true
@@exit_false:
    mov eax, 0
    ret
@@exit_true:
    mov eax, 1
    ret
AboutDialog endp
end start
```

Окна диалога, имеющие большое количество элементов управления, должны, соответственно, иметь диалоговые процедуры, обрабатывающие сообщения от этих элементов. В нашем примере определены два окна диалога, внешне очень похожие друг на друга. С их помощью пользователь может задавать координаты начала и конца отрезка и углов прямоугольника. На рис. 16.3 приведен вид окна приложения с окном диалога для задания координат прямоугольника.

При задании координат пользователь сам должен контролировать правильность ввода данных, так как алгоритм программы не предусматривает их проверки. Пра-

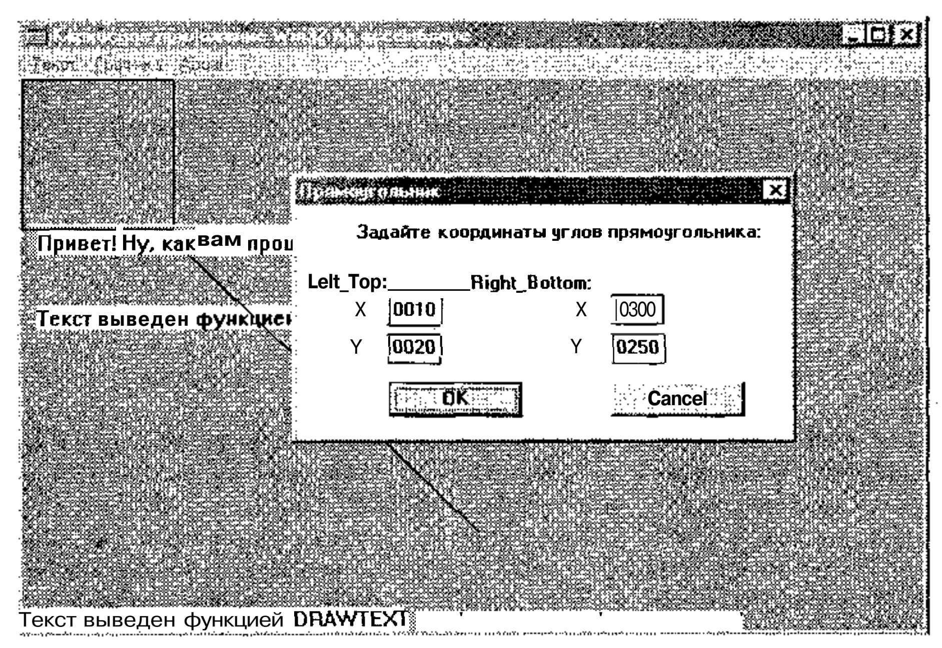


Рис. 16.3. Окно диалога для задания координат прямоугольника

вильные значения, вводимые в каждое из полей ввода, должны содержать все четыре десятичные цифры с включением, при необходимости, ведущих нулей (!).

Для обработки сообщений от этих окон диалога программа prg16.4.asm (см. листинг 16.13) содержит две диалоговые процедуры, DialogProc1 и DialogProc2. Процедура DialogProc1 (строки 357-439) предназначена для ввода координат отрезка, процедура DialogProc2 (строки 440-522) — координат прямоугольника. Структура диалоговой процедуры аналогична структуре оконной процедуры. В начале диалоговой процедуры находится код, определяющий тип поступившего сообщения и в зависимости от него передающий управление в определенную точку диалоговой процедуры:

```
mov eax, @@message
cmp ax, WM_INITDIALOG
je  @@wminitdialog
cmp ax, WM_COMMAND
jne @@exit_false
mov ebx, @@wparam     ;в bx идентификатор элемента управления
cmp bx, IDOK
je  @@idok
cmp bx, IDCANCEL
je  @@idcancel
jmp @@exit_false
```

Главное сообщение, поступающее в оконную процедуру, — сообщение WM_COMMAND. Именно оно несет информацию о действиях пользователя с элементами управления окна диалога. Так, в строках 361-371 листинга 16.13, обрабатываются два сообщения: WM_INITDIALOG и WM_COMMAND. Сообщение WM_INITDIALOG приходит в диалоговую процедуру один раз. Это происходит в процессе инициализации окна диалога перед его появлением в окне приложения. Далее в диалого-

вую процедуру поступает последовательность сообщений WM_COMMAND. Параметр 'wParam такого сообщения содержит идентификатор того элемента управления, над которым пользователь произвел некоторое действие, например, щелкнул на кнопке OK или Cancel (см. рис. 16.3). Обратите внимание на то, что при описании элементов различных окон диалога используются одни и те же идентификаторы. Более того, в файле prg16_4.h этим идентификаторам назначены одни и те же константы. Именно их значения передаются в структуре сообщения для идентификации конкретного элемента окна диалога. Одинаковые значения не вносят никакой путаницы в работу приложения, так как для каждого окна диалога существует своя оконная процедура.

Интерес представляют детали реализации диалоговых процедур DialogProcl и DialogProc2. В них используются две отладочные макрокоманды, sim4_to_ EAXbin и show_eax.

Макрокоманда sim4_to_EAXbin в программе prg16_4.asm предназначена для преобразования строки из четырех символов десятичных цифр в эквивалентное двоичное число, помещаемое в регистр EAX. В данной программе эта макрокоманда требуется для преобразования символьных строк, считываемых из полей ввода окна диалога. Эти символьные строки логически представляют собой координаты отрезка и прямоугольника. Преобразованные с помощью макрокоманды sim4_to_EAXbin значения этих координат используются функциями Win32 API MoveToEx (строки 272-277), LineTo (строки 278-282) и Rectangle (строки 297-316). В конце своей работы макрокоманда sim4_to_EAXbin с помощью окна, выводимого функцией MessageBox, сообщает о результатах преобразования. Это отладочный момент работы, и, учитывая учебный характер разрабатываемого приложения, он не убран из его конечной версии.

Макрокоманда show_eax также используется для отладки и является аналогом макрокоманды show, описанной в главе 14. Попытка запустить макрокоманду show в нашем приложении ни к чему хорошему не приведет, поэтому для динамического контроля за работой программы макрокоманда show_eax может оказаться очень полезной. Кстати, проблема отладки Windows-приложений довольно актуальна, и для ее решения подчас приходится применять самые нетрадиционные подходы, но это уже тема для отдельного разговора.

Активизация окна диалога

Для того чтобы отобразить окно диалога на экране и выполнить с его помощью некоторую работу, используется функция DialogBoxParamA. Эта функция имеет следующий формат (в нотации C/C++):

```
int DialogBoxParam (HINSTANCE hInstance,//дескриптор приложения
    LPCT5TR lpTemplateName, //указатель на строку с заголовком окна
    HWND hWndParent,//дескриптор окна
    DLGPROC lpDialogFunc,  //указатель на диалоговую процедуру
    LPARAM dwInitParam) //значение, передаваемое в диалоговую функцию
                        //через lParam
```

В программе prg16_4.asm (см. листинг 16.13) обращение к этой функции производится трижды при выборе соответствующих пунктов меню (строки 271, 296 и 344 в теле процедуры MenuProc). С ее помощью в диалоговую процедуру передается 32-разрядное значение, которое извлекается посредством параметра IParam.

По окончании работы окно диалога должно быть закрыто функцией EndDialog, имеющей формат

```
BOOL EndDialog (HWND hDlg, //дескриптор окна диалога
   int nResult); //возвращаемое значение
```

Обычно закрытие окна производится щелчком на кнопке OK или Cancel.

Большой размер исходного текста приложения prg16_4.asm может кого-то испугать. Он может быть существенно сокращен за счет различных средств ассемблера, в том числе макрокоманд и процедур. Если с их помощью реорганизовать исходный текст, то он станет гораздо читабельнее и удобнее для использования, почти как аналогичный текст на языке C/C++. Однако в отличие от текста на C/C++ текст на ассемблере — готовая машинная программа, в которую ничего лишнего не добавляется и код которой полностью контролируется. Более того, *с* этим текстом можно проводить весь комплекс мероприятий по его оптимизации. В нашем примере намеренно ничего не сделано в плане улучшения внешнего вида исходного текста, чтобы не навязывать читателю свои подходы к этому процессу, тем более что это могло бы ввести его в заблуждение. Текст программы в настоящем виде более всего отражает суть процессов, происходящих в системе Windows во время исполнения приложения, и механизм взаимодействия приложения с системой.

Нами остались не разработанными варианты реакции приложения на выбор пользователем пунктов меню Графика ▶ Эффекты ▶ Павлин и Графика ▶ Эффекты ▶ Кружева. Их реализация требует использования команд сопроцессора и поэтому будет рассмотрена в следующей главе.

В начале главы отмечалось, что Windows поддерживает работу двух типов приложений — оконных, в полной мере использующих все достоинства графического интерфейса, и консольных, работающих в текстовом режиме. Большая часть данной главы была посвящена разработке программ с оконным интерфейсом. Заключительную часть главы посвятим рассмотрению основ программирования консольных приложений.

Программирование консольных Windows-приложений

Язык ассемблера — язык системных программистов, исследователей принципов работы операционных систем, программ и аппаратных средств. Здесь не всегда нужны красивые графические оболочки, а наоборот, велика потребность в удобных средствах для работы с текстовой информацией. Операционная система Windows обеспечивает встроенную поддержку *консолей,* которые, по определению, являются интерфейсами ввода-вывода для приложений, работающих в текстовом режиме. Понятие «консоль» существует в вычислительной технике давно. В общем случае под «консолью» подразумевают текстовый терминал для управления компьютером. Видимая часть такого терминала — клавиатура (для ввода управляющих воздействий) и монитор (как средство отображения-реакции вычислительной системы). В Windows консоль представляет собой приложение, которое позволяет взаимодействовать с операционной системой посредством ввода текстовых

команд. Такой способ управления компьютером позволяет решать в основном административные задачи. Приложение, с помощью которого поддерживается этот режим, называется *консольным*. Видимая часть консольных приложений называется *окном консольного приложения*.

Написание консольных приложений на ассемблере — задача более актуальная, чем написание оконных. Причина простая — малыми затратами нам становятся доступны практически все возможности Win32 API. Программист может запустить одновременно нескольких консольных приложений и при этом работать с мышью и клавиатурой в стиле Windows. Далее мы рассмотрим порядок действий для запуска консольного Windows-приложения и организацию обмена данными с ним.

API Win32 предоставляет два разных уровня работы с консолью — высокий и низкий. Выбор нужного уровня зависит от того, какая степень гибкости и полноты контроля требуется приложению для обеспечения своей работы с консолью. Функции высокого уровня обеспечивают простоту процесса ввода-вывода за счет использования стандартных дескрипторов ввода-вывода, но при этом невозможен доступ к входному и экранным буферам консоли. Функции низкого уровня требуют учета большего количества деталей и написания большего объема кода, но это компенсируется большей гибкостью.

Консоль состоит из одного входного и нескольких экранных буферов. Входной буфер представляет собой очередь, каждая запись которой содержит информацию относительно отдельного входного события консоли. Экранный буфер — двухмерный массив, содержащий символы, выводимые в окно консоли, и данные об их цвете.

Очередь входного буфера содержит информацию о следующих событиях:

- нажатии и отпускании клавиш;
- манипуляциях мышью — движение, нажатие и отпускание кнопок;
- изменении размера активного экранного буфера, состоянии прокрутки.

С каждой консолью связаны две кодовые таблицы — по одной для ввода и вывода. Консоль использует входную кодовую таблицу для трансляции ввода с клавиатуры в соответствующие символьные значения. Аналогичным образом используется кодовая таблица вывода — для трансляции символьных значений, формируемых различными функциями вывода, в символы, отображаемые в окне консоли. Для работы с кодовыми таблицами приложение может задействовать пары функций: SetConsoleCP и GetConsoleCP — для входных кодовых таблиц и SetConsoteOutputCP и GetConsoleOutputCP — для выходных кодовых таблиц. Идентификаторы кодовых таблиц, доступные на данном компьютере, сохраняются в системном реестре следующим ключом:

```
HKEY_LOCAL_MACHINE\\SYSTEM\\CurrentControlSet\\Control\\Nls\\CodePage
```

Для поддержки консольных приложений Win32 API содержит более сорока функций, предназначенных для интеграции в среду Windows программ, работающих в текстовом режиме. Данные функции обеспечивают поддержку отмеченных ранее двух уровней доступа к консоли — высокого и низкого. Консольные функции ввода высокого уровня позволяют приложению извлечь данные, полученные при вводе с клавиатуры и сохраненные во входном буфере консоли. Консольные функции вывода высокого уровня позволяют приложению записать данные в уст-

ройство стандартного вывода или в устройство ошибки с тем, чтобы отобразить этот текст в экранном буфере консоли. Функции высокого уровня также поддерживают переназначение стандартных дескрипторов ввода-вывода и управление режимами работы консоли. Консольные функции низкого уровня позволяют приложениям получить детальную информацию о вводе с клавиатуры, событиях нажатия и отпускания кнопок мыши и о манипуляциях пользователя с окном консоли. Все это обеспечивает высокую степень контроля над выводом данных на экран. Высокоуровневый и низкоуровневый консольный ввод-вывод не являются взаимоисключающими, и приложение может использовать любую комбинацию этих функций.

Минимальная программа консольного приложения

Минимальная программа консольного приложения на ассемблере выглядит так, как показано в листинге 16.14.

Листинг 16.14. Фрагменты приложения prg16_5.asm

```
;prg16_5.asm
;Пример минимальной программы консольного Windows-приложения
.486
.model flat,STDCALL      ;модель памяти flat
include WindowConA.inc
;Объявление внешними используемых в данной программе функций Win32 (ASCII):
extrn   AllocConsole:PROC
extrn   SetConsoleTitleA:PROC
extrn   ExitProcess:PROC
.data
TitleText    db   'Win32-console application',0
.code
start   proc near       ;точка входа в программу:
;запрос  консоли
    call     AllocConsole
;проверить успех запроса консоли
    test     eax,eax
    jz  exit            ;неудача
;выведем заголовок окна консоли SetConsoleTitle:
    push     offset TitleText
    call     SetConsoleTitleA
;проверить успех вывода заголовка
    test     eax,eax
    jz  exit            ;неудача
;-----------------------------------------------
;работаем ...
;-----------------------------------------------
exit:                   ;выход из приложения
;готовим вызов VOID ExitProcess(UINT uExitCode)
    push     0
    call     ExitProcess
start   endp
end start
```

Если убрать комментарии, то кода будет совсем немного. В нем представлены вызовы трех функций: AllocConsole, SetConsoleTitle, ExitProcess.

Первой функцией консольного приложения должна быть функция запроса консоли AllocConsole:

```
BOOL AllocConsole(VOID);
```

Для вызова функции AllocConsole не требуется никаких параметров. В случае успеха функция AllocConsole возвращает ненулевое значение, при неудаче — нуль. Выделенная консоль представляет собой типичное для Windows окно. Процесс в конкретный момент времени может использовать одну консоль. Если ему нужно запустить еще одну консоль, то прежняя должна быть закрыта или освобождена с помощью функции FreeConsole:

```
BOOL FreeConsole(VOID);
```

В случае успеха функция FreeConsole возвращает ненулевое значение, при неудаче — нуль.

При завершении процесса выделенная процессу консоль освобождается автоматически. В нашем случае использован именно этот вариант закрытия консоли — функцией ExitProcess:

```
VOID ExitProcess(UINT uExitCode);
```

Функции ExitProcess передается код завершения процесса и всех завершаемых цепочек в этом процессе. Проанализировать этот код можно с помощью функций GetExitCodeProcess и GetExitCodeThread. В общем случае в различных ветвях кода может быть несколько точек выхода с вызовом функции ExitProcess. Задавая различные значения кода завершения, можно идентифицировать причину завершения процесса.

Окно консоли может иметь заголовок, для отображения которого предназначена функция SetConsoleTitle:

```
BOOL SetConsoleTitle(LPCTSTR lpConsoleTitle);
```

У функции SetConsoleTitle один параметр — указатель на строку с заголовком консоли, заканчивающуюся нулем.

Организация высокоуровневого консольного ввода-вывода

Для высокоуровневого ввода-вывода приложение может использовать файловые функции ReadFile и WriteFile, а также функции консольного ввода-вывода ReadConsole и WriteConsole. Эти функции обеспечивают косвенный доступ к входному и экранным буферам пульта. Физически эти функции фильтруют записи входного буфера консоли так, чтобы возвратить ввод как поток символов, игнорируя все другие записи с расширенной информацией о мыши, клавиатуре и изменении размеров окна консоли. Отфильтрованный поток символов отображается в окне консоли, начиная с текущей позиции курсора. Существуют два важных различия в использовании пар функций ReadFile\WriteFile и ReadConsole\WriteConsole.

- Поддержка символов Unicode и ANSI. Консольные функции (ReadConsole\WriteConsole) поддерживают эти наборы, а файловые (ReadFile\WriteFile) — нет.
- Функции файлового ввода-вывода могут использоваться для обращения как к файлам, так и к именованным каналам (устройствам, присоединенным к последовательному интерфейсу). Консольные функции ввода-вывода можно использовать только с тремя дескрипторами стандартного ввода-вывода (см. далее описание функций ReadConsole\WriteConsole).

Из сказанного ранее следует, что функции высокоуровневого ввода-вывода обеспечивают простой способ обмена (чтения-записи) потоков символов с консолью.

Операция чтения высокого уровня реализуется функцией ReadConsole, которая получает входные символы из буфера ввода консоли и сохраняет их в указанном буфере:

```
BOOL ReadConsole(HANDLE hConsoleInput, LPVOID lpBuffer,
    DWORD nNumberOfCharsToRead, LPDWORD lpNumberOfCharsRead,
    LPVOID lpReserved);
```

Параметры этой функции означают следующее:

- hConsolelnput — дескриптор входного потока консоли;
- lpBuffer — указатель на строку, в которую будет записана вводимая строка символов;
- nNumberOfCharsToRead — размер буфера, указанного lpBuffer;
- lpNumberOfCharsRead — количество действительно введенных символов;
- lpReserved — этот параметр не используется, поэтому должен задаваться как NULL

Операция записи высокого уровня реализуется функцией WriteConsole, которая извлекает символы из указанного буфера и записывает их в экранный буфер, начиная с текущей позиции курсора и продвигая ее по мере записи символов.

```
BOOL WriteConsole(HANDLE hConsoleOutput, CONST VOID *lpBuffer,
    DWORD nNumberOfCharsToWrite, LPDWORD lpNumberOfCharsWritten,
    LPVOID lpReserved);
```

Параметры этой функции означают следующее:

- hConsoleOutput — дескриптор выходного потока консоли;
- lpBuffer — указатель на выводимую строку;
- nNumberOfCharsToWrite — размер буфера, указанного lpBuffer;
- lpNumberOfCharsWritten — количество действительно выведенных символов;
- lpReserved — этот параметр не используется, поэтому должен задаваться как NULL.

Для своей работы эти и некоторые другие консольные функции требуют получения стандартных дескрипторов ввода-вывода. Значения этих дескрипторов присваиваются параметрам hConsolelnput и hConsoleOutput. По умолчанию стандартный дескриптор ввода связан с клавиатурой, стандартный дескриптор вывода — с экраном. Получить стандартный дескриптор ввода-вывода можно с помощью функции GetStdHandle.

```
HANDLE GetStdHandle(DWORD nStdHandle);
```

На вход функции GetStdHandle должно быть подано одно из следующих значений:

- STD_INPUT_HANDLE = -10 — дескриптор стандартного входного потока;
- STD_OUTPUT_HANDLE = -11 — дескриптор стандартного выходного потока;
- STD_ERROR_HANDLE = -12 — дескриптор стандартного потока ошибок.

Используя функции высокоуровневого ввода-вывода, приложение может управлять цветами текста и фона, которыми должны отображаться символы, записываемые в экранный буфер. Приложению доступны следующие функции высокоуровневого консольного ввода-вывода:

- эхо-контроль вводимых символов на экране из активного экранного буфера;

- ввод строки, окончание операции чтения которой происходит при нажатии клавиши Enter;
- автоматическая обработка некоторых символов, вводимых с клавиатуры: перевода каретки, нажатия клавиш Ctrl+C и т. д.;
- автоматическая обработка некоторых символов, выводимых на экран: перевода строки и каретки, возврата на один символ и т. д.

Функция SetConsoleCursorPosition предназначена для указания позиции, с которой начинается выполнение операций чтения-записи в окно консоли:

```
BOOL SetConsoleCursorPosition(HANDLE hConsoleOutput,
     COORD dwCursorPosition);
```

Параметрами этой функции являются стандартный дескриптор вывода hConsoleOutput, полученный функцией GetStdHandle, и указатель на структуру COORD с координатами новой позиции курсора:

```
COORD    struc
x    dw  0
y    dw  0
     ends
```

По умолчанию цветовое оформление окна консоли довольно унылое — черный фон, белый текст. Внести разнообразие во внешний вид окна консоли поможет функция SetConsoleTextAttribute, с помощью которой можно изменить цвета, установленные по умолчанию для текста и фона:

```
BOOL SetConsoleTextAttribute(HANDLE hConsoleOutput, WORD wAttributes);
```

Первый параметр — без комментариев, второй определяет цвет текста и фона. Второй параметр формируется через операцию логического ИЛИ следующих значений:

- FOREGROUND_BLUE = 0001h - синий текст;
- FOREGROUND_GREEN = 0002h - зеленый текст;
- FOREGROUND_RED = 0004h - красный текст;
- FOREGROUND_INTENSITY = 0008h - текст повышенной яркости;
- BACKGROUND_BLUE = 0010h - голубой фон;
- BACKGROUND_GREEN = 0020h - зеленый фон;
- BACKGROUND_RED = 0040h - красный фон;
- BACKGROUND_INTENSITY = 0080h - фон повышенной яркости.

Для задания белого цвета складываются три цветовых компонента, для задания черного компоненты не задаются вовсе.

Пример программы консольного ввода-вывода

Для демонстрации функций высокоуровневого ввода-вывода в окно консоли разработаем программу, которая вводит с клавиатуры строку и отображает ее в заголовке окна консоли, а затем выводит эту строку в окне консоли с изменением текущей позиции курсора и цвета текста (листинг 16.15).

Листинг 16.15. Фрагменты приложения prg16_6.asm

```
;prg16_6.asm - программа консольного ввода-вывода
;с изменением атрибутов выводимого текста
;...
```

```
.data
;...
.code
start   proc near         ;точка входа в программу:
;...
;получим стандартные дескрипторы ввода-вывода
    push    STD_OUTPUT_HANDLE
    call    GetStdHandle
    mov     dOut,eax          ;dOut - дескриптор консольного вывода
    push    STD_INPUT_HANDLE
    call    GetStdHandle
    mov     dIn,eax           ;dIn - дескриптор консольного ввода
;введем строку
;установим курсор в позицию (2,6)
    mov     con.xx,2
    mov     con.yy,6
    push    con
    push    dOut
    call    SetConsoleCursorPosition
    cmp     eax,0
    jz      exit              ;если неуспех
    push    0
    push    offset NumWri     ;количество действительно введенных символов
    push    80                ;размер буфера TitleText для ввода
    push    offset TitleText
    push    dIn
    call    ReadConsoleA
    cmp     eax,0
    jz      exit              ;если неуспех
;выведем введенную строку в заголовок окна консоли:
    push    offset TitleText
    call    SetConsoleTitleA
;проверить успех вывода заголовка
    test    eax,eax
    jz      exit              ;неудача
;выведем строку в окно консоли с различных позиций и с разными цветами
;установим курсор в позицию (2,5)
    mov     ecx,10            ;строку выведем 10 раз
    mov     bl,10000001b      ;начальные атрибуты
ml: push    ecx
    inc     con.xx
    inc     con.yy
    push    con
    push    dOut
    call    SetConsoleCursorPosition
    cmp     eax,0
    jz      exit              ;если неуспех
;определим атрибуты выводимых символов:
;будем получать их циклически сдвигом - регистр BL
    xor     eax,eax
    rol     bl,1
    mov     al,bl
    push    eax
    push    dOut
    call    SetConsoleTextAttribute
    cmp     eax,0
    jz      exit              ;если неуспех
;вывести строку
    push    0
    push    offset NumWri     ;реальное количество выведенных на экран
символов
    push    NumWri            ;длина строки для вывода на экран
    push    offset TitleText;адрес строки для вывода на экран
    push    dOut
    call    WriteConsoleA
```

продолжение ↗

Листинг 16.15 *(продолжение)*

```
    cmp     eax,0
    jz      exit            ;если неуспех
    pop     ecx
    loopml
exit:                       ;выход из приложения
;...
```

Каждый консольный процесс имеет собственный список функций-обработчиков, которые вызываются системой, когда происходят определенные события, например, при активном окне консоли пользователь нажимает комбинацию клавиш Ctrl+C, Ctrl+Break или Ctrl+Close. При запуске консольного приложения список функций-обработчиков содержит только заданную по умолчанию функцию, которая вызывает функцию ExitProcess. Консольный процесс может добавлять или удалять дополнительные функции-обработчики, вызывая функцию SetConsoleCtrlHandler:

```
BOOL SetConsoleCtrlHandler(PHANDLER_ROUTINE HandlerRoutine, BOOL Add);
```

Данная функция имеет два параметра:

- HandlerRoutine — указатель на определенную приложением функцию HandlerRoutine, которая должна быть добавлена или удалена;
- Add — логическое значение:
  - 1 — функция должна быть добавлена;
  - 0 — функцию необходимо удалить.

Функция HandlerRoutine — это определенная приложением функция обратного вызова. Консольный процесс использует эту функцию, чтобы обработать нажатия клавиш управления. На самом деле HandlerRoutine — идентификатор-заполнитель для определенного приложением имени функции:

```
BOOL WINAPI HandlerRoutine(DWORD dwCtrlType);
```

Параметр DwCtrlType определяет тип сигнала управления, получаемого обработчиком. Этот параметр может принимать одно из следующих значений:

- CTRL_C_EVENT = 0 — сигнал, имитирующий нажатие клавиш Ctrl+C, может быть получен из двух источников: с клавиатуры или от функции GenerateConsoleCtrlEvent;
- CTRL_BREAK_EVENT = 1 — сигнал, имитирующий нажатие клавиш Ctrl+Break, может быть получен из двух источников: с клавиатуры или от функции GenerateConsoleCtrlEvent;
- CTRL_CLOSE_EVENT = 2 — сигнал, который система посылает всем процессам, подключенным к данному консольному приложению, когда пользователь его закрывает (выбирая пункт Close в системном меню окна консоли или щелкая на кнопке завершения задачи в диалоговом окне Менеджера задач);
- CTRL_LOGOFF_EVENT = 5 — сигнал, который посылается всем консольным процессам, когда пользователь завершает работу в системе (этот сигнал не указывает, какой именно пользователь завершает работу);
- CTRL_SHUTDOWN_EVENT = 6 — сигнал, который система посылает всем консольным процессам при подготовке к выключению машины.

Функция HandlerRoutine должна возвратить логическое значение: 1 — если она обрабатывает конкретный сигнал управления; 0 — для обработки полученного события будет использоваться другая функция-обработчик HandlerRoutine из списка функций-обработчиков этого процесса (то есть включенная в этот список раньше данной функции).

Как уже было упомянуто, каждый консольный процесс может определить несколько функций HandlerRoutine, которые связываются в цепочку. Первоначально этот список содержит только заданную по умолчанию функцию-обработчик, вызывающую функцию ExitProcess, что в результате приводит к завершению текущего консольного приложения. Консольный процесс добавляет или удаляет дополнительные функции-обработчики, вызывая функцию SetConsoleCtrlHandler, которая не затрагивает список функций-обработчиков других процессов. Когда консольный процесс принимает любой из сигналов управления (см. выше), то вызывается последняя зарегистрированная функция-обработчик, если она не возвращает 1, то управление передается следующему (предыдущему) зарегистрированному обработчику и так продолжается до тех пор, пока один из обработчиков не возвратит 1. Если ни один из обработчиков этого не сделал, то вызывается обработчик, заданный по умолчанию.

Установка обработчиков для сигналов CTRL_CLOSE_EVENT, CTRL_LOGOFF_EVENT и CTRL_SHUTDOWN_EVENT дает процессу возможность выполнить специфичные для него действия по корректному завершению приложения. Пользовательская функция HandlerRoutine может быть вызвана для того, чтобы выполнить следующие действия:

- вызвать функцию ExitProcess для завершения процесса;
- возвратить 0 (ложь) — это означает, что завершение приложения должен выполнить обработчик, заданный по умолчанию;
- возвратить 1 — в этом случае никакие другие функции-обработчики не вызываются, и система отображает модальное окно диалога с запросом о необходимости завершения процесса.

В последнем случае система также отображает окно диалога, если процесс не отвечает определенное время (5 секунд для CTRL_CLOSE_EVENT и 20 секунд для CTRL_LOGOFF_EVENT и CTRL_SHUTDOWN_EVENT). Процесс может использовать функцию SetProcessShutdownParameters, чтобы запретить системе отображать окно диалога. В этом случае система просто заканчивает процесс, когда HandlerRoutine возвращает истину или когда истекает определенный период времени.

В листинге 16.16 приведен пример пользовательского обработчика события — ввода комбинации Ctrl+C или Ctrl+Break. За основу взята предыдущая программа.

Листинг 16.16. Фрагменты приложения prg16_7.asm

```
;prg16_7.asm - программа, демонстрирующая
;пользовательский обработчик события.
;...
.data
;...
Text_CTRL_C db  "Нажаты CTRL+C"
Len_Text_CTRL=$-Text_CTRL_C
Text_BREAK      db  "Нажаты CTRL+BREAK"
```

продолжение ↗

Листинг 16.16 *(продолжение)*

```
Len_BREAK=$-Text_BREAK
.code
CtrlHandler proc
arg @@dwCtrlType:DWORD
uses ebx,edi, esi            ;эти регистры обязательно должны сохраняться
;анализируем тип сигнала управления
    cmp @@dwCtrlType,CTRL_C_EVENT
    je  h_CTRL_C_EVENT
    cmp @@dwCtrlType,CTRL_BREAK_EVENT
    je  h_CTRL_BREAK_EVENT
    jmp h_default
h_CTRL_C_EVENT:              ;при нажатии CTRL+C выводим сообщение:
;установим курсор
;...
    call    SetConsoleCursorPosition
;...
;вывести строку Text_CTRL_C
;...
    call    WriteConsoleA
;...
;возвращаем признак обработки
    mov eax,1
    jmp exit_CtrlHandler
h_CTRL_BREAK_EVENT:          ;при нажатии CTRL+BREAK выводим сообщение:
;установим курсор
;...
    call    SetConsoleCursorPosition
;...
;вывести строку
;...
    call    WriteConsoleA
;...
;возвращаем признак обработки
    mov eax,1
    jmp exit_CtrlHandler
h_default:  mov eax,0ffffffffh  ;возвращаем -1
                                ;все остальное не обрабатываем
exit_CtrlHandler:   ret
CtrlHandler endp
start   proc near            ;точка входа в программу:
;работаем...
;получим стандартные дескрипторы ввода-вывода
;...
;установим функцию-обработчик сигналов управления
    push    TRUE
    push    offset cs: CtrlHandler
    call    SetConsoleCtrlHandler
    cmp     eax, 0
    jz      exit             ;если неуспех
;введем строку в буфер TitleText
;установим курсор в позицию (2,6)
;...
    call    SetConsoleCursorPosition
;...
    call    ReadConsoleA
;...
;выведем введенную строку в заголовок окна консоли:
    push    offset TitleText
    call    SetConsoleTitleA
;...
;выведем строку в окно консоли с различных позиций и с разными цветами
    mov     ecx,10           ;строку выведем 10 раз
    mov     bl,10000001b     ;начальные атрибуты
```

```
m1: push    ecx
;установим курсор в позицию
;...
    call    SetConsoleCursorPosition
;...
;определим атрибуты выводимых символов:
;будем получать их циклически - сдвигом регистра BL
    xor eax,eax
    rol bl,1
    mov al,bl
    push    eax
    push    dOut
    call    SetConsoleTextAttribute
;...
;вывести строку TitleText
;...
    call    WriteConsoleA
    cmp eax,0
    jz  exit                ;если неуспех
    pop ecx
    loop    m1
;...
```

Относительно этой программы можно сделать два замечания. Первое касается функции HandlerRoutine, которая в нашей программе называется CtrlHandler. Как упоминалось, эта функция является функцией обратного вызова. Она вызывается при возникновении определенных событий неявно — из системы Windows. По структуре и алгоритму работы она аналогична оконной функции, которую мы рассматривали ранее (см. раздел «Каркасное Windows-приложение на C/C++»). Поэтому за всеми подробностями отсылаем читателя к этому материалу. Второе замечание касается порядка отладки приложений, содержащих определяемые пользователем функции (процедуры) обратного вызова. Первое, что нужно сделать в процессе пошагового выполнения программы в отладчике, — выяснить адрес процедуры обратного вызова. В программе из листинга 16.16 это можно сделать, выяснив, какое значение будет помещено в стек при выполнении команд:

```
;...
;установим функцию-обработчик сигналов управления
    push    TRUE
    push    offset cs: CtrlHandler
    call    SetConsoleCtrlHandler
    cmp     eax, 0
    jz      exit    ;если неуспех
;...
```

После этого, сделав активным окно CPU отладчика (выбрав в меню команду View ▸ CPU), необходимо установить указатель мыши в окно с командами процессора и щелкнуть правой кнопкой мыши. В появившемся контекстном меню выберите пункт Goto. В результате этих действий отладчик отобразит окно диалога, в которое необходимо внести адрес программы-обработчика CtrlHandler. После этого в верхней части окна команд отобразится первая команда процедуры CtrlHandler. Установите на нее курсор и нажмите клавишу F4. Программа начнет выполняться по своему алгоритму. При нажатии пользователем управляющих комбинаций клавиш, допустимых функцией HandlerRoutine, управление будет передано этой функции, и вы сможете произвести ее отладку.

Низкий уровень консольного ввода-вывода по сравнению с высоким обладает более широкими и гибкими возможностями. Низкоуровневые функции консоль-

ного ввода-вывода обеспечивают прямой доступ к входному и экранным буферам консоли, предоставляя приложению доступ к событиям мыши и клавиатуры, а также к информации об изменении размеров окна консоли. Функции низкоуровневого ввода-вывода позволяют приложению иметь доступ по чтению-записи к указанному числу последовательных символьных ячеек в экранном буфере или к прямоугольному блоку символьных ячеек в указанной позиции экранного буфера.

Итоги

- Разработка Windows-приложения на языке ассемблера — вполне реальное и в ряде случаев оправданное дело. Однако несмотря на имеющиеся в TASM и MASM средства, для создания полноценного Windows-приложения требуются дополнительные программные и информационные ресурсы, предоставляемые пакетами языков высокого уровня. Лучше всего для этой цели подходит пакет VC++ версии 6.0 и выше. Основную ценность в нем имеют включаемые файлы, редактор ресурсов, работающий в составе интегрированной среды разработки, и компилятор ресурсов. Интерес могут представлять также различные утилиты, входящие в состав пакета Visual C++, например Spy++.
- Приступать к разработке приложения для системы Windows на ассемблере лучше всего, имея некоторый опыт разработки приложений на языке высокого уровня. Это необходимо для понимания логики работы приложения. Когда понимание логики работы Windows-приложения достигнуто, выбор языка для его реализации приобретает в большей степени техническое значение и определяется постановкой задачи и предполагаемыми условиями ее эксплуатации.
- Благодаря поддержке системой Windows консольных приложений можно малыми силами решать серьезные задачи, в том числе и по администрированию системы. Консольным приложениям доступны практически все возможности, предоставляемые Win32 API, при этом для построения приложений не требуется реализовывать какие-то изощренные схемы.

Глава 17
Архитектура и программирование сопроцессора

- Архитектура
- Форматы данных
- Система команд
- Исключения и их обработка
- Использование отладчика

В предыдущих главах мы рассматривали команды и алгоритмы обработки целочисленных данных, то есть чисел с фиксированной точкой (см. главы 5 и 8). Для обработки числовых данных в формате с плавающей точкой процессоры IA-32 содержат специальное устройство, которое является важной частью их архитектуры.

Устройства для обработки чисел с плавающей точкой появились в компьютерах давно. С точки зрения архитектуры они выглядели по-разному. Так, в архитектуре ЕС ЭВМ (аналог IBM 360/370) устройство с плавающей точкой было естественной частью этой архитектуры со своим регистровым пространством и подсистемой команд. Архитектура компьютеров на базе процессоров вначале опиралась исключительно на целочисленную арифметику. С ростом мощи, а главное, с осознанием разработчиками процессорной техники того факта, что их устройства могут составить достойную конкуренцию своим «большим» предшественникам, в архитектуре компьютеров на базе процессоров стали появляться устройства для обработки чисел с плавающей точкой. В архитектуре семейства процессоров Intel 80x86 устройство для обработки чисел с плавающей точкой появилось

в составе компьютера на базе процессора i8086/88 и получило название *математический сопроцессор* (далее просто *сопроцессор).* Выбор такого названия был обусловлен тем, что, во-первых, это устройство было предназначено для расширения вычислительных возможностей основного процессора, а во-вторых, оно было реализовано в виде отдельной микросхемы, то есть его присутствие было необязательным. Микросхема сопроцессора для процессора i8086/88 имела название i8087. С появлением новых моделей процессоров Intel совершенствовались и сопроцессоры, хотя их программная модель осталась практически неизменной. Как отдельные (а соответственно, необязательные в конкретной комплектации компьютера) устройства сопроцессоры сохранялись вплоть до модели процессора i386 и имели название i287 и i387 соответственно. Начиная с модели i486 сопроцессор исполняется в одном корпусе с основным процессором и, таким образом, является неотъемлемой частью компьютера.

Для чего нужен сопроцессор, какие возможности добавляет он к тому, что делает основной процессор, кроме обработки еще одного формата данных? Перечислим некоторые из них.

- Полная поддержка стандартов IEEE-754 и 854 на арифметику с плавающей точкой. Эти стандарты описывают как форматы данных, с которыми должен работать сопроцессор, так и набор реализуемых им функций.
- Поддержка численных алгоритмов для вычисления значений тригонометрических функций, логарифмов и т. п. Эта работа сопроцессора выполняется абсолютно прозрачно для программиста, что само по себе очень ценно, так как не требует от него разработки соответствующих подпрограмм.
- Обработка десятичных чисел с точностью до 18 разрядов, что позволяет сопроцессору без округления выполнять арифметические операции над целыми десятичными числами со значениями до 10^{18}.
- Обработка вещественных чисел из диапазона $3{,}37 \cdot 10^{-4932}...1{,}18 \cdot 10^{+4932}$.

Нужно отметить, что в последние годы разработчики компьютерной периферии все активнее освобождают центральный процессор от части вычислений с целью более эффективной реализации специализированных операций. Очень ярко это проявляется на рынке мультимедийного оборудования, где множество разработчиков предлагают видеокарты с чипсетами (наборами микросхем), более эффективно реализующими работу с графикой, чем это делает сам процессор. Несмотря на это дополнение системы целочисленных команд процессора командами сопроцессора предоставляет ряд уникальных свойств и возможностей, пренебрегать которыми было бы опрометчиво.

Архитектура сопроцессора

В главе 2 мы определили место сопроцессора в архитектуре компьютера. Аппаратная реализация сопроцессора нас интересует лишь в видимой для программиста части. Как и в случае с основным процессором, интерес для нас представляет *программная модель сопроцессора.* С точки зрения программиста, сопроцессор представляет собой совокупность регистров, каждый из которых имеет свое функциональное назначение (рис. 17.1).

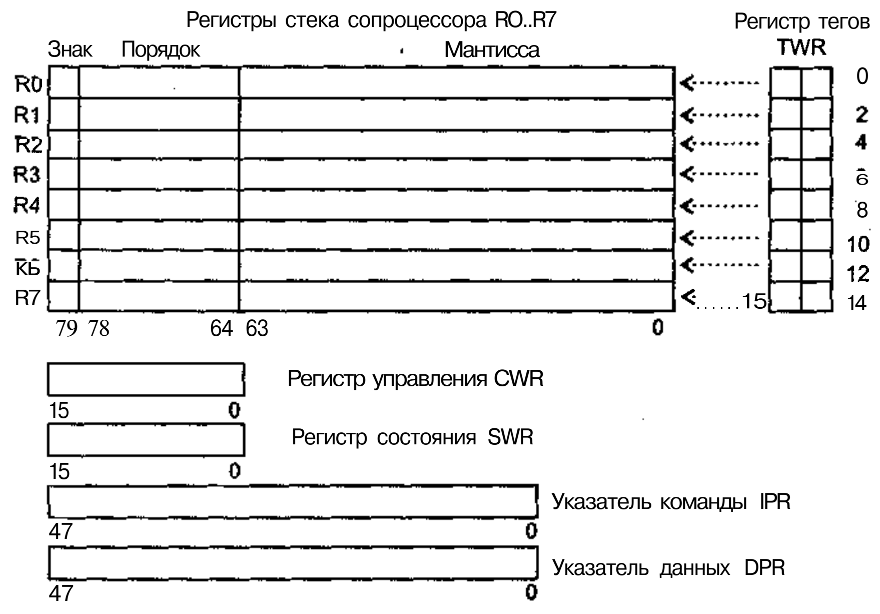


Рис. 17.1. Программная модель сопроцессора

В программной модели сопроцессора можно выделить три группы регистров.

- Восемь регистров R0...R7 составляют основу программной модели сопроцессора — *стек сопроцессора*. Размерность каждого регистра — 80 битов. Такая организация характерна для устройств, специализирующихся на обработке вычислительных алгоритмов. Вспомните, как представляются математические выражения с использованием обратной польской записи (ПОЛИЗ). Вычисление такого выражения заключается в выборке с вершины стека очередной операции. Если это двухместная операция, то с вершины стека снимаются два операнда, над которыми и производятся действия в соответствии со снятой ранее операцией. Более подробно представление выражения в форме ПОЛИЗ мы рассмотрим далее. Реализация численных алгоритмов на основе стека сопроцессора позволяет получить существенный выигрыш в скорости вычислений.
- Три служебных регистра:
  - *регистр состояния сопроцессора* SWR (Status Word Register) отражает информацию о текущем состоянии сопроцессора и содержит поля, позволяющие определить, какой регистр является текущей вершиной стека сопроцессора, какие исключения возникли после выполнения последней команды, каковы особенности выполнения последней команды (некий аналог регистра флагов основного процессора) и т. д.;
  - *управляющий регистр сопроцессора* CWR (Control Word Register) управляет режимами работы сопроцессора; с помощью полей в этом регистре можно регулировать точность выполнения численных вычислений, управлять округлением, маскировать исключения;

- *регистр слова тегов* TWR (Tags Word Register) используется для контроля за состоянием каждого из регистров R0...R7 (команды сопроцессора используют этот регистр, например, для того, чтобы определить возможность записи значений в указанные регистры).

- Два регистра указателей — данных DPR (Data Point Register) и команд IPR (Instruction Point Register) — предназначены для запоминания информации об адресе команды, вызвавшей исключительную ситуацию, и адресе ее операнда. Эти указатели используются при обработке исключительных ситуаций (но не для всех команд).

Все эти регистры являются программно доступными. Однако к одним из них доступ получить довольно легко, для этого в системе команд сопроцессора существуют специальные команды, а к другим его получить сложнее, так как специальных команд для этого нет, поэтому необходимо выполнять дополнительные действия.

Рассмотрим общую логику работы сопроцессора и более подробно охарактеризуем перечисленные регистры.

Регистровый стек сопроцессора организован по принципу кольца. Это означает, что среди всех регистров, составляющих стек, нет такого, который является вершиной стека. Напротив, все регистры стека с функциональной точки зрения абсолютно равноправны. Но, как известно, в стеке всегда должна быть вершина. И она действительно есть, но является плавающей. Контроль текущей вершины осуществляется аппаратно с помощью трехразрядного поля TOP регистра SWR (рис. 17.2). В поле TOP фиксируется номер регистра стека 0...7 (R0...R7), являющегося текущей вершиной стека.

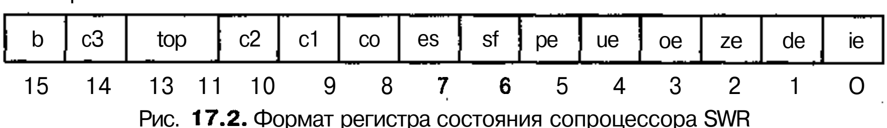
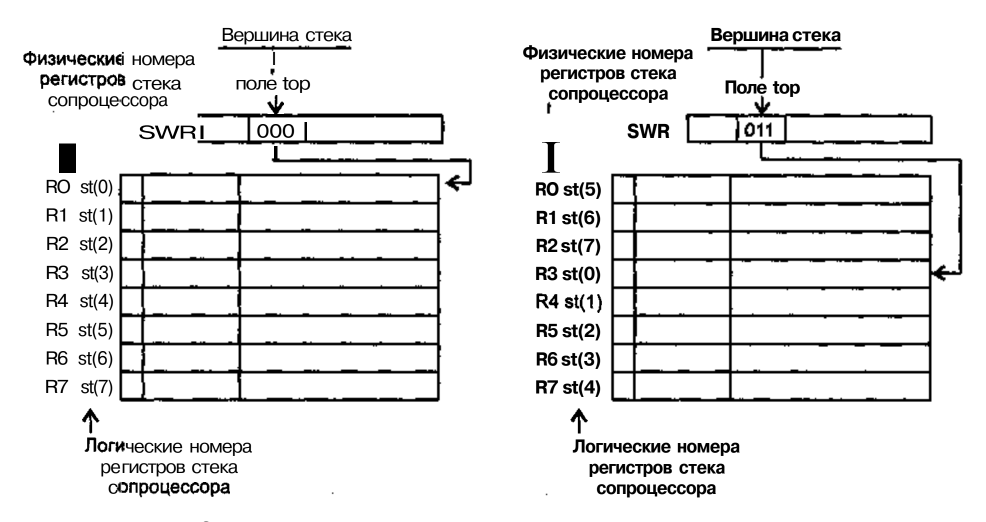


Рис. **17.2.** Формат регистра состояния сопроцессора SWR

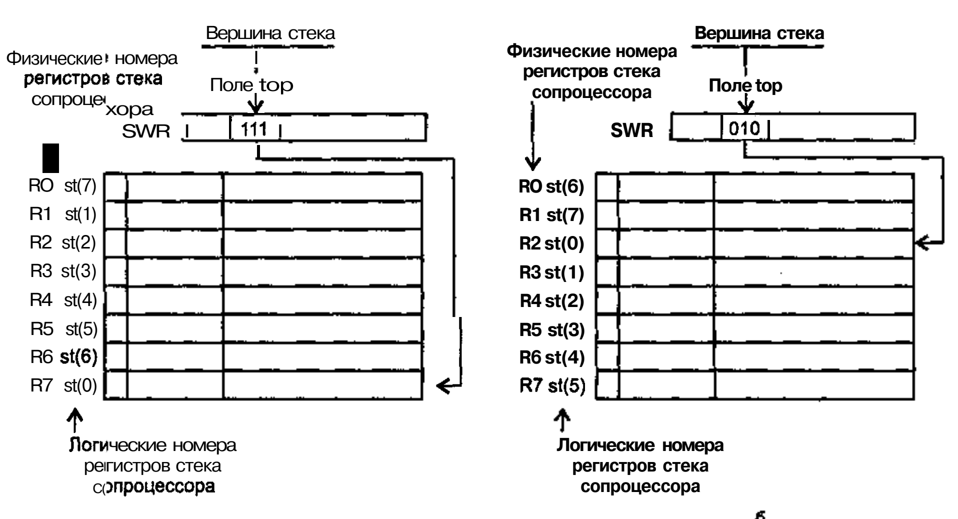
Команды сопроцессора не оперируют физическими номерами регистров стека RO...R7. Вместо этого они используют логические номера этих регистров ST(0)... ST(1). С помощью логических номеров реализуется относительная адресация регистров стека сопроцессора. Если текущей вершиной стека является физический регистр RO, то после записи очередного значения в стек сопроцессора его текущей вершиной станет физический регистр R7 (рис. 17.3, *а*). На рис. 17.3, б показан пример, когда текущей вершиной до записи в стек является физический регистр R3, а после записи в стек текущей вершиной становится физический регистр стека R2. То есть по мере записи в стек указатель его вершины движется по направлению к младшим номерам физических регистров (уменьшается на единицу). Что касается логических номеров регистров стека ST(0)...ST(1), то, как следует из рисунка, они «плавают» вместе с изменением текущей вершины стека. Таким образом, реализуется принцип кольца.

На первый взгляд, такая организация стека кажется странной. Но, как оказалось, она обладает большой гибкостью. Это хорошо видно на примере передачи

Состояние стека сопроцессора до выполнения операции записи в стек



Состояние стека сопроцессора после выполнения операции записи в стек



а б

Рис. 17.3. Физическая и логическая нумерации регистров стека сопроцессора

параметров подпрограмме. Для повышения гибкости подпрограмм (в разработке и использовании) нежелательно привязывать их по передаваемым параметрам к аппаратным ресурсам (физическим номерам регистров сопроцессора). Гораздо удобнее задавать порядок следования передаваемых параметров в виде логических номеров, поскольку такой способ передачи однозначен и не требует от разработчика знания лишних подробностей о вариантах аппаратной реализации сопроцессора. Логическая нумерация регистров сопроцессора, поддерживаемая на уровне системы команд, идеально реализует эту идею. При этом не имеет значения, в какой физический регистр стека сопроцессора были помещены данные перед вызовом

подпрограммы, определяющим является только порядок следования параметров в стеке. По этой причине подпрограмме достаточно знать не место, а только порядок размещения передаваемых параметров в стеке.

Перед тем как приступить к описанию команд и данных, с которыми работает сопроцессор, отметим, каким образом «уживаются» между собой эти два разных вычислительных устройства — процессор и сопроцессор. Каждое из них имеет свои несовместимые друг с другом системы команд и форматы обрабатываемых данных. Несмотря на то что сопроцессор архитектурно представляет собой отдельное вычислительное устройство, он не может существовать отдельно от основного процессора. Ранее уже было отмечено, что первые модели процессоров Intel (i8086, i286, i386) и сопроцессоров (соответственно, i8087, i287, i387) выполнялись как отдельные устройства — сопроцессор при необходимости вставлялся в специальный разъем на системной плате. Начиная с модели i486 сопроцессор и основной процессор производятся в одном корпусе и являются физически неделимыми, то есть архитектурно это по-прежнему два разных устройства, а аппаратно — одно.

Процессор и сопроцессор, являясь двумя самостоятельными вычислительными устройствами, могут работать параллельно. Но этот параллелизм касается только их внутренней работы над исполнением очередной команды. Как реализован этот параллелизм, в чем его суть и особенности? Оба процессора подключены к общей системной шине и имеют доступ к одинаковой информации. Инициирует процесс выборки очередной команды всегда основной процессор. После выборки команда попадает одновременно в оба процессора. Любая команда сопроцессора имеет код операции, первые 5 битов которого имеют значение 11011. Когда код операции начинается этими битами, то основной процессор по дальнейшему содержимому кода операции выясняет, требует ли данная команда обращения к памяти. Если это так, то основной процессор формирует физический адрес операнда и обращается к памяти, после чего содержимое ячейки памяти выставляется на шину данных. Если обращение к памяти не требуется, то основной процессор заканчивает работу над данной командой (не делая попытки ее исполнения!) и приступает к декодированию следующей команды из текущего входного командного потока. Что же касается сопроцессора, то выбранная команда, как уже было отмечено, попадает к нему и к основному процессору одновременно. Сопроцессор, определив по первым пяти битам, что очередная команда принадлежит его системе команд, начинает ее исполнение. Если команда требует операнда в памяти, то сопроцессор обращается к шине данных за чтением содержимого ячейки памяти, которое к этому моменту должно быть предоставлено основным процессором. Из этой схемы взаимодействия следует, что в определенных случаях необходимо согласовывать работу обоих устройств. К примеру, если во входном потоке сразу за командой сопроцессора следует команда основного процессора, использующая результаты работы предыдущей команды, то сопроцессор не успеет выполнить свою команду за то время, пока основной процессор, пропустив сопроцессорную команду, выполняет свою. Очевидно, что логика работы программы нарушается. Возможна и другая ситуация. Если входной поток команд содержит последовательность из нескольких команд сопроцессора, то, очевидно, что процессор, в отличие от сопроцессора, проверит их очень быстро, чего он не должен делать, так как обеспечивает внешний интерфейс для сопроцессора. Эти и другие более сложные ситуации при-

водят к необходимости синхронизировать между собой работу двух процессоров. В первых моделях процессоров подобная синхронизация проводилась программистом «вручную» путем вставки перед или после каждой команды сопроцессора специальной команды WAIT или FWAIT. Работа данной команды заключалась в приостановке работы основного процессора до тех пор, пока сопроцессор не закончит работу над последней командой. Начиная с модели процессора i486, подобная синхронизация выполняется командами WAIT/FWAIT, которые введены в алгоритм работы большинства команд сопроцессора, и программисту нет необходимость заботиться об этом явно. Но некоторые команды из группы команд управления сопроцессором (см. далее) имеют два варианта реализации — с синхронизацией (ожиданием) и без нее.

Из всего сказанного можно сделать важный вывод: использование сопроцессора является совершенно прозрачным для программиста. В общем случае программисту следует воспринимать сопроцессор как набор дополнительных регистров, для работы с которыми предназначены специальные команды. Для эффективного применения сопроцессора программист должен хорошо разобраться в структуре регистров и логике их использования. Поэтому перед тем как приступить к рассмотрению команд и данных, с которыми работает сопроцессор, приведем описание структуры некоторых регистров сопроцессора.

Регистр состояния SWR

Ранее было отмечено, что регистр SWR отражает текущее состояние сопроцессора после выполнения последней команды. Далее перечислены поля, из которых структурно состоит регистр SWR (см. рис. 17.2).

- Шесть флагов исключительных ситуаций.
- Бит SF (Stack Fault) — ошибка работы стека сопроцессора. Бит устанавливается в единицу, если возникает одна из трех исключительных ситуаций (см. ранее) — PE, UE или IE. В частности, его установка информирует о попытке записи в заполненный стек или, напротив, попытке чтения из пустого стека. После того как вы проанализировали этот бит, его нужно снова установить в ноль вместе с битами PE, UE или IE (если они были установлены);
- Бит ES (Error Summary) сигнализирует о суммарной ошибке в работе сопроцессора. Бит устанавливается в единицу, если возникает любая из шести исключительных ситуаций, о которых будет рассказано далее.
- Четыре бита C0...C3 (Condition Code) представляют собой код условия. Назначение этих битов аналогично флагам в регистре EFLAGS основного процессора — они отражают результат выполнения последней команды сопроцессора. В приложении А для некоторых команд сопроцессора приведена интерпретация битов C0...C3.
- Трехразрядное поле TOP содержит указатель регистра текущей вершины стека.

Почти половину регистра SWR занимают биты (флаги) регистрации исключительных ситуаций. *Исключительная ситуация* — особый тип прерываний. Необходимо заметить [8], что прерывания, поддерживаемые процессором Intel, по месту их возникновения классифицируются на внешние и внутренние. Внутренние

прерывания возникают в ходе работы текущей программы и делятся на синхронные (по команде int) и асинхронные, называемые *исключениями,* или *особыми случаями.* Таким образом, *исключения* — это разновидность прерываний, с помощью которых процессор информирует программу о некоторых особенностях ее реального исполнения. Сопроцессор также обладает способностью возбуждения подобных прерываний при возникновении определенных ситуаций (не обязательно ошибочных). Все возможные исключения сведены к шести типам, каждому из которых соответствует один бит в регистре SWR. Программисту совсем не обязательно писать обработчик для реакции на ситуацию, приведшую к некоторому исключению. Сопроцессор умеет самостоятельно реагировать на многие из них. Это так называемая обработка исключений по умолчанию. Для того чтобы запретить сопроцессору обработку определенного типа исключения по умолчанию, необходимо это исключение замаскировать. Такое действие выполняется путем установки в единицу нужного бита в управляющем регистре сопроцессора CWR (рис. 17.4). Приведем типы исключений, фиксируемые с помощью регистра SWR:

- IE (Invalide operation Error) — недействительная операция;
- DE (Denormalized operand Error) — денормализованный операнд;
- ZE (divide by Zero Error) — ошибка деления на нуль;
- OE (Overflow Error) — ошибка переполнения (возникает в случае выхода порядка числа за максимально допустимый диапазон);
- UE (Underflow Error) — ошибка антипереполнения (возникает, когда результат слишком мал);
- PE (Precision Error) — ошибка точности (устанавливается, когда сопроцессору приходится округлять результат из-за того, что его точное представление невозможно; так, сопроцессору, как и читателю, никогда не удастся точно разделить 10 на 3).

При возникновении любого из этих шести типов исключений устанавливается в единицу соответствующий бит в регистре SWR вне зависимости от того, было ли замаскировано это исключение в регистре CWR или нет. Более подробно об исключениях, в частности, об условиях их возникновения, рассказывается в разделе «Исключения сопроцессора и их обработка».

Регистр управления CWR

Регистр управления сопроцессором CWR определяет особенности обработки численных данных (рис. 17.4). Он состоит:

- из шести масок исключений;
- поля управления точностью PC (Precision Control);
- поля управления округлением RC (Rounding Control).

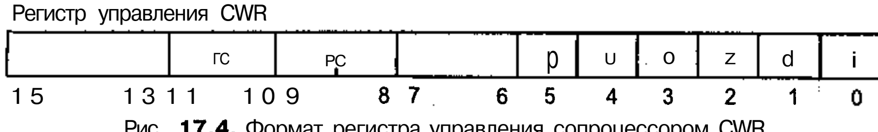


Рис. 17.4. Формат регистра управления сопроцессором CWR

Шесть масок предназначены для маскирования исключительных ситуаций, возникновение которых фиксируется с помощью шести битов регистра SWR. Если какие-то биты исключений в регистре CWR установлены в единицу, это означает, что соответствующие исключения будут обрабатываться самим сопроцессором. Если для какого-либо исключения в соответствующем бите масок исключений регистра CWR содержится нулевое значение, то при возникновении исключения этого типа будет возбуждено прерывание 16 (10h). Операционная система должна содержать (или программист должен написать) обработчик этого прерывания. Он должен выяснить причину прерывания, после чего, если это необходимо, устранить ее, а также выполнить другие действия. Более подробно этот вопрос обсуждается в разделе «Исключения сопроцессора и их обработка».

Поле управления точностью PC предназначено для выбора длины мантиссы. Возможные значения в этом поле означают:

- PC = 00 — длина мантиссы 24 бита;
- PC = 10 — длина мантиссы 53 бита;
- PC = 11 — длина мантиссы 64 бита.

По умолчанию устанавливается значение поля PC - 11.

Поле RC позволяет управлять процессом округления чисел в ходе работы сопроцессора. Необходимость округления может возникнуть в ситуации, когда после выполнения очередной команды сопроцессора получается непредставимый результат, например периодическая дробь 3,333... Установив одно из значений в поле RC, можно выполнить округление в необходимую сторону. Для того чтобы выяснить характер округления, введем обозначения:

- *m* — значение в ST(0) или результат работы некоторой команды, который не может быть точно представлен и поэтому должен быть округлен;
- *a* и *b* — наиболее близкие значения к значению *m*, которые могут быть представлены в регистре ST(0) сопроцессора, причем выполняется условие $a < m < b$.

Далее приведены значения поля RC и описан соответствующий им характер округления:

- 00 — значение *m* округляется к ближайшему числу *a* или *b*;
- 01 — значение *m* округляется в меньшую сторону, то есть $m = a$;
- 10 — значение *m* округляется в большую сторону, то есть $m = b$;
- 11 — производится отбрасывание дробной части *m* (может использоваться в операциях целочисленной арифметики).

Регистр тегов TWR

Регистр тегов TWR представляет собой совокупность двухразрядных полей. Каждое двухразрядное поле соответствует определенному физическому регистру стека (см. рис. 17.1) и характеризует его текущее состояние. Изменение состояния любого регистра стека отражается на содержимом соответствующего этому регистру поля регистра тега. Возможны следующие значения в полях регистра тега:

- 00 — регистр стека сопроцессора занят допустимым ненулевым значением;
- 01 — регистр стека сопроцессора содержит нулевое значение;

- 10 — регистр стека сопроцессора содержит одно из специальных численных значений (см. ниже), за исключением нуля;
- 11 — регистр пуст, и в него можно производить запись (нужно отметить, что это значение в одном из двухразрядных полей регистра тегов не означает, что все биты соответствующего регистра стека обязательно нулевые).

Мы не раз уже отмечали, что при написании программы разработчик манипулирует не абсолютными, а относительными номерами регистров стека. По этой причине у него могут возникнуть трудности при попытке интерпретации содержимого регистра тегов TWR с соответствующими физическими регистрами стека. В качестве связующего звена необходимо привлекать информацию из поля TOP регистра SWR.

Форматы данных

Сопроцессор расширяет номенклатуру форматов данных, с которыми работает основной процессор. В этом нет ничего удивительного, так как формат данных любого устройства в существенной мере отражает специфику его работы. Сопроцессор специально разрабатывался для вычислений с плавающей точкой. Но сопроцессор может работать и с целыми числами, хотя и менее эффективно. Перечислим форматы данных, с которыми работает сопроцессор:

- двоичные целые числа в трех форматах — 16, 32 и 64 бита;
- упакованные целые десятичные (BCD) числа — длина максимального числа составляет 18 упакованных десятичных цифр (9 байтов);
- вещественные числа в трех форматах — коротком (32 бита), длинном (64 бита), расширенном (80 битов).

Кроме этих основных форматов, сопроцессор поддерживает специальные численные значения, к которым относятся:

- денормализованные вещественные числа — это числа, меньшие минимального нормализованного числа (см. ниже) для каждого вещественного формата, поддерживаемого сопроцессором;
- нуль;
- положительные и отрицательные значения бесконечности;
- нечисла;
- неопределенности и неподдерживаемые форматы.

Рассмотрим более подробно основные форматы данных, поддерживаемые сопроцессором. Важно отметить, что в самом сопроцессоре числа в этих форматах имеют одинаковое внутреннее представление — расширенный формат вещественного числа. Это один из форматов представления вещественных чисел, который точно соответствует формату регистров R0...R7 стека сопроцессора (см. рис. 17.1). Таким образом, даже если вы используете команды сопроцессора с целочисленными операндами, то после загрузки в сопроцессор операндов целого типа они автоматически преобразуются в формат расширенного вещественного числа.

Двоичные целые числа

Сопроцессор работает с тремя типами целых чисел (рис. 17.5).

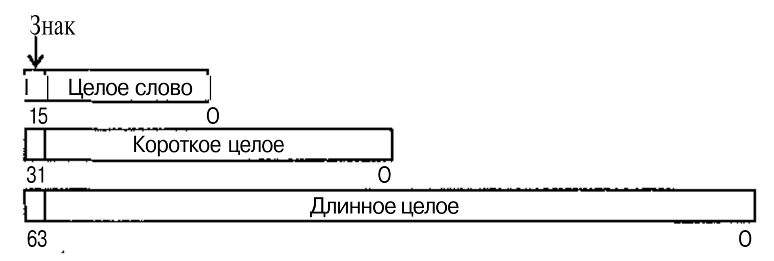


Рис. **17.5.** Форматы целых чисел сопроцессора

В табл. 17.1 представлены формат целых чисел, их размерность и диапазон значений.

Таблица 17.1. Форматы целых чисел сопроцессора

Формат	Размер, битов	Диапазон значений
Целое слово	16	-32 768...+32 767
Короткое целое	32	$-2 \cdot 10^9...+2 \cdot 10^9$
Длинное целое	64	$-9 \cdot 10^{18}...+9 \cdot 10^{18}$

Выбирая формат данных, с которыми будет работать ваша программа, помните, что сопроцессор поддерживает операции с целыми числами, но работа с ними осуществляется неэффективно. Причина в том, что обработка сопроцессором целочисленных данных будет замедлена из-за дополнительного преобразования целых чисел в их внутреннее представление в виде эквивалентного вещественного числа расширенного формата.

В программе целые двоичные числа описываются обычным способом — с использованием директив DW, DD и DQ. Например, целое число 5 может быть описано следующим образом:

```
ch_dw   5   ;представление в памяти: ch_dw=05 00
ch_dd   5   ;представление в памяти: ch_dw=05 00 00 00
ch_dq   5   ;представление в памяти: ch_dw=05 00 00 00 00 00 00 00
```

Работать с целыми числами может далеко не всякая команда сопроцессора. Подробную информацию о командах сопроцессора можно найти в приложении.

Упакованные целые десятичные числа

Сопроцессор поддерживает один формат упакованных целых десятичных чисел, или BCD-чисел (рис. 17.6). Как вы помните, для описания упакованного десятичного числа используется директива DT (см. главу 8). Данная директива позволяет описать 20 цифр в упакованном десятичном числе (по две в каждом байте). Из-за того что максимальная длина упакованного десятичного числа в сопроцессоре составляет только 9 байт, в регистры RO...R7 можно поместить только 18 упакованных десятичных цифр. Старший десятый байт игнорируется. Самый старший бит этого байта используется для хранения знака числа.

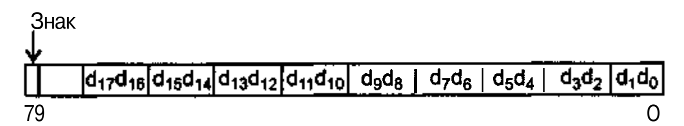


Рис. **17.6.** Формат десятичного числа сопроцессора

Упакованные десятичные числа также представляются в стеке сопроцессора в расширенном формате. Упакованные десятичные числа в программе описываются директивой DT. Например, целое число 5 365 904 в формате упакованного десятичного числа может быть описано следующим образом:

```
ch_dt    dt   5365904
;представление в памяти: ch_dt=04 59 36 05 00 00 00 00 00 00
```

Нужно отметить, что в сопроцессоре имеются всего две команды для работы с упакованными десятичными числами — это команды сохранения и загрузки.

Вещественные числа

Основной тип данных, с которыми работает сопроцессор, — вещественный. Данные этого типа описываются тремя форматами: коротким, длинным и расширенным (рис. 17.7).

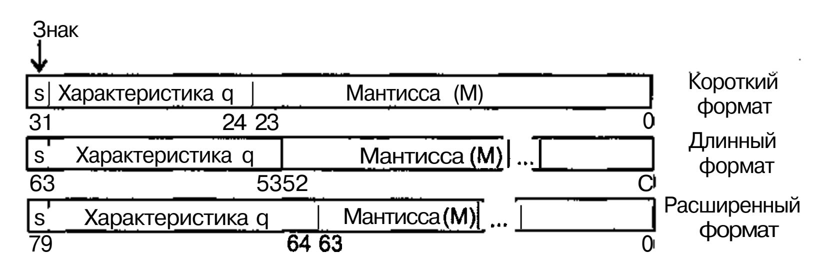


Рис. 17.7. Форматы вещественных чисел сопроцессора

Для представления вещественного числа используется следующая формула:

$$A - (\pm M) \cdot N^{\pm(p)} \qquad (17.1)$$

Здесь:

- M — мантисса числа A (мантисса должна удовлетворять условию $|M| < 1$);
- N — основание системы счисления, представленное целым положительным числом;
- p — порядок числа, показывающий истинное положение точки в разрядах мантиссы (по этой причине вещественные числа имеют еще название чисел с плавающей точкой, так как ее положение в разрядах мантиссы зависит от значения порядка).

Для удобства обработки в процессоре чисел с плавающей точкой его архитектурой накладываются некоторые ограничения на компоненты формулы (17.1). Далее перечислены эти условия и ограничения для сопроцессоров, применяющихся в архитектуре IA-32.

- Основание системы счисления $N = 2$.
- Мантисса M должна быть представлена в *нормализованном* виде. Нормализация может отличаться для разных типов процессоров. Для ЕС ЭВМ, например, мантисса нормализованного числа должна удовлетворять условию $1/N \leq |M| < 1$. Это означает, что старший бит представления должен быть единичным. Для случая, когда $N = 2$, это соответствует отношению $1/2 < |M| < 1$ или в двоичном виде $0{,}10...00 < |M| < 0{,}11...11$, то есть первая цифра после запятой должна быть значащей (единицей), а порядок p, соответственно, таким, чтобы это условие выполнялось. Для архитектуры сопроцессора IA-32 нормализованным является число несколько иного вида:

$$Л = (-1)^s \cdot N^q \cdot M \tag{17.2}$$

Здесь:

- s — значение знакового разряда (0 — число больше нуля, 1 — число меньше нуля);
- q — порядок числа, его назначение аналогично назначению порядка p в формуле (17.1), но как поясняется далее, p и q — не одно и тоже.

В этой формуле знак имеют и порядок вещественного числа, и его мантисса. На рис. 17.7 видно, что формат хранения вещественного числа в памяти имеет только поле для знака мантиссы. А где же хранится знак порядка?

В сопроцессоре Intel на аппаратном уровне принято соглашение, что порядок p определяется в формате вещественного числа особым значением q, называемым *характеристикой*. Величина q связана с порядком p посредством следующей формулы и представляет собой некоторую константу (условно назовем ее *фиксированным смещением*):

$$q = p + \text{фиксированное смещение.} \tag{17.3}$$

Для каждого из трех возможных форматов вещественных чисел смещение q имеет разное, но фиксированное для конкретного формата значение, которое зависит от количества разрядов, отводимых под характеристику (табл. 17.2).

Таблица 17.2. Форматы вещественных чисел

Формат	Короткий	Длинный	Расширенный
Длина числа (биты)	32	64	80
Размерность мантиссы M	24	53	64
Диапазон значений	$10^{-38}...10^{+38}$	$10^{-308}...10^{+308}$	$10^{-4932}...10^{+4932}$
Размерность характеристики q	8	И	15
Значение фиксированного смещения	+127	+1023	+16 383
Диапазон характеристик q	0...255	0...2047	0...32 767
Диапазон порядков p	-126...+127	-1022...+1023	–16 382...+16 383

В таблице показаны диапазоны значений характеристик q и соответствующих им истинных порядков p вещественных чисел. Отметим, что нулевому порядку вещественного числа в коротком формате соответствует значение характеристики

равное 127, которому в двоичном представлении соответствует значение 01 11 11 11. Отрицательному порядку p, например -1, будет соответствовать характеристика $q = -1 + 127 = 126$, или в двоичном виде — 01 11 11 10. Положительному порядку p, например +1, будет соответствовать характеристика $q = 1 + 127 = 128$, или в двоичном виде — 10 00 00 00. То есть все положительные порядки имеют в двоичном представлении характеристики старший бит равный единице, а отрицательные порядки — нет. Таким образом, знак порядка «спрятан» в старшем бите характеристики. Теперь вам должно быть понятно, откуда появились значения в двух последних строках табл. 17.2.

Так как нормализованное вещественное число всегда имеет целую единичную часть (исключая перечисленные ранее специальные численные значения), то при его представлении в памяти появляется возможность считать первый разряд вещественного числа единичным по умолчанию и учитывать его наличие только на аппаратном уровне. Это дает возможность увеличить диапазон представимых чисел, так как появляется лишний разряд, пригодный для представления мантиссы числа. Но это справедливо только для короткого и длинного форматов вещественных чисел. Расширенный формат как внутренний формат представления числа любого типа в сопроцессоре содержит целую единичную часть вещественного в явном виде.

Как определить вещественное число или зарезервировать место для его размещения в программе на ассемблере?

Короткое вещественное число длиной в 32 разряда определяется директивой DD. При этом обязательным в записи числа является наличие десятичной точки, даже если оно не имеет дробной части. Для транслятора десятичная точка является указанием, что число нужно представить в виде числа с плавающей точкой в коротком формате (см. рис. 17.7). Это же касается длинного и расширенного форматов представления вещественных чисел, определяемых директивами DQ и DT. Другой способ задания вещественного числа директивами DD, DQ и DT — экспоненциальная форма с использованием символа «e». Вид вещественного числа в поле операндов директив DD, DQ и DT можно представить синтаксической диаграммой (рис. 17.8).

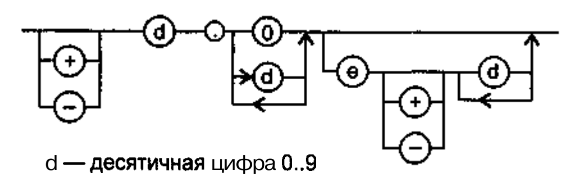


Рис. 17.8. Синтаксис вещественных чисел в директивах DD, DQ и DT

Чтобы окончательно разобраться в тонкостях представления вещественных чисел различных форматов в памяти, рассмотрим несколько примеров.

Определим в программе вещественное число 45,56 в коротком формате. Это можно сделать тремя способами:

```
dd 45.56
dd 45.56e0
dd 0.4556e2
```

В памяти это число будет выглядеть так:

71 3d 36 42.

Учитывая, что в архитектуре Intel принят «перевернутый» порядок следования байтов в памяти в соответствии с принципом «младший байт по младшему адресу», истинное представление числа 45,56 будет следующим:

42 36 3d 71.

Двоичное представление в памяти числа 45,56 иллюстрирует рис. 17.9.

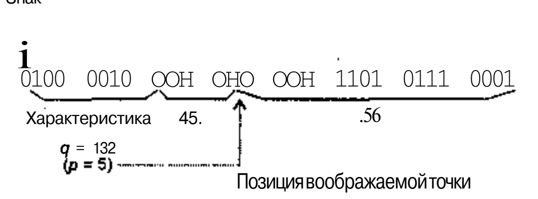


Рис. 17.9. Двоичное представление в памяти вещественного числа в директиве DD

Из рисунка видно, что старшая единица мантиссы при представлении в памяти отсутствует.

Определим теперь в программе вещественное число 45,56 в длинном формате. Это можно сделать двумя способами:

```
dq  45.56
dq  45.56e0
```

В памяти это число будет выглядеть так:

47 e1 7a 14 ae c7 46 40.

Перевернув его, получим истинное значение:

40 46 c7 ae 1 4 7a e1 47.

Разберите его по компонентам вещественного числа самостоятельно.

Наконец, определим в программе вещественное число 45,56 в расширенном формате:

```
dt  45.56
```

В памяти это число будет выглядеть так:

71 3d 0a d7 a3 70 3d b6 04 40.

Перевернув его, получим истинное значение в памяти:

40 04 b6 3d 70 a3 d7 0a 3d 71.

Двоичное представление числа 45,56 полезно рассмотреть подробнее (рис. 17.10):

Знак мантиссы

01000000000000100101101100011110101110000101000111101011100001010111000010111000 1

Характеристика q = 16 388 (p = 5) — Позиция воображаемой точки — .56

Рис. 17.10. Двоичное представление в памяти вещественного числа в директиве DT

0100 0000 0000 0100 1011 0110 0011 1101 0111 0000 1010
OOH 1101 0111 0000 1010 11100001 0111 0001.

Данное число имеет следующее назначение битов:

- 0 — знак«+»;
- 100 0000 0000 0100 - характеристика q = 16 388;
- 1011 01 — целая часть числа (45);
- 10 0011 1101 0111 0000 1010 0011 1101 0111 0000 1010 1110 0001 0111 0001 - дробная часть числа (0,56).

Как видно, в мантиссе явно присутствует старшая единица, чего не было в коротком и длинном форматах представления вещественного числа.

Дальнейшее обсуждение требует четкого понимания того, каким образом из дробного десятичного числа получается его расширенное вещественное представление. Рассмотрим этот процесс по шагам.

1. Переведем десятичную дробь 45,56 в двоичное представление. Подробно алгоритм такого перевода описан в главе 4. В результате получим эквивалентное двоичное представление:

$$45{,}56_{10} =$$
$$= 101101.1000111101011100001010001111010111000010101110000101110001.$$

2. Нормализуем число. Для этого переносим точку влево до тех пор, пока в целой части числа не останется одна двоичная единица. Число переносов влево (или вправо, если десятичное число было меньше единицы) станет порядком числа. Но будьте внимательны — в поле характеристики заносится смещенное значение порядка (см. табл. 17.2). Таким образом, после перемещения точки получаем значение порядка равное 5. Соответственно, характеристика будет выглядеть так:

$$5 + 16\,383 - 16\,388_{10} = 100000000000100_2$$

Сформированный результат в виде вещественного числа в расширенном формате состоит из трех компонентов:

- знака — 0;
- характеристики — 100000000000100;
- мантиссы - 1011 01100011 1101 0111 0000 1010 0011 1101 0111 0000 1010 1110 0001 0111 0001.

Позже вы научитесь пользоваться отладчиком для работы с сопроцессором и получите возможность просматривать содержимое регистров стека (см. раздел «Использование отладчика»). При этом содержимое регистров стека будет изображаться в шестнадцатеричном виде: 40 04 b6 3d 70 a3 d7 Oa 3d 71. Видно, что это представление полностью совпадает с приведенным ранее представлением числа в памяти, если оно описано в директиве DT.

В качестве итога еще раз подчеркнем, что расширенный формат представления вещественного числа является единственным форматом представления чисел в регистрах сопроцессора. Само преобразование производится автоматически при загрузке числа в стек сопроцессора. Исключение составляет расширенный формат.

Специальные численные значения

Несмотря на большой диапазон вещественных значений, представимых в регистрах стека сопроцессора, понятно, что бесконечные значения находятся за рамками этого диапазона. Для того чтобы иметь возможность реагировать на вычислительные ситуации, в которых возникают такие значения, в сопроцессоре и предусмотрены специальные комбинации битов, называемые *специальными численными значениями.* При необходимости программист может сам кодировать специальные численные значения, поскольку вещественные числа, описанные директивой DT, соответствующие команды сопроцессора загружают без всяких преобразований.

Денормализованные вещественные числа

Денормализованные вещественные числа — это числа, которые меньше минимального нормализованного числа для каждого вещественного формата. Поясним природу денормализованных чисел с использованием числовой шкалы. Например, для вещественного числа в расширенном формате диапазон представимых значений в сопроцессоре на числовой шкале будет выглядеть так, как показано на рис. 17.11.

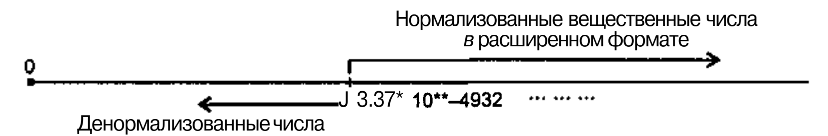


Рис. 17.11. Положение денормализованных вещественных чисел на числовой шкале

Как нам уже известно, сопроцессор хранит числа в нормализованном виде. По мере приближения чисел к нулю ему все труднее «вытягивать» их значения к нормализованному виду, то есть к такому виду, чтобы первой значащей цифрой мантиссы была единица. Размерность разрядной сетки, отведенной в форматах вещественных чисел сопроцессора для представления характеристики, не безгранична. Поэтому при определенных значениях числа в расширенном формате значение характеристики становится равным нулю (рис. 17.11). Но на самом деле число отлично от нуля, то есть это «не настоящий» численный нуль. Таким образом, между

Минимальное положительное денормализованное число:

0	00..00	00000.........001
79	78 64	63 0

Минимальное отрицательное денормализованное число:

1	00..00	0000.........001
79	78 64	63 0

Максимальное положительное денормализованное число:

0	00..00	11111.........111
79	78 64	63 0

Максимальное отрицательное денормализованное число:

1	00..00	1111.........111
79	78 64	63 0

Рис. 17.12. Диапазон представимых в сопроцессоре денормализованных чисел

истинным нулем и минимально представимым нормализованным числом есть еще бесконечное количество очень маленьких чисел. Это и есть так называемые денормализованные числа. Они имеют нулевой порядок и ненулевую мантиссу. Диапазон представимых в сопроцессоре денормализованных чисел не безграничен, так как количество разрядов мантиссы ограниченно (рис. 17.12).

Вопрос о том, каким образом сопроцессор реагирует на появление денормализованных чисел, будет рассмотрен в разделе «Исключения сопроцессора и их обработка». При формировании денормализованного значения в некотором регистре стека в соответствующем этому регистру теге регистра TWR формируется специальное значение (10).

Нуль

Нуль также относят к специальным численным значениям. Это делается из-за того, что это значение особо выделяется среди корректных вещественных значений, формируемых как результат работы некоторой команды. Более того, нуль может формироваться как реакция сопроцессора на определенную вычислительную ситуацию.

Значение истинного нуля может иметь знак (рис. 17.13), что, впрочем, не влияет на его восприятие командами сопроцессора. Если необходимо определить знак нуля, то используйте команду FXAM. В результате работы этой команды в бит C1 регистра SWR заносится знак операнда. При загрузке нуля в регистр стека в соответствующем теге регистра TWR формируется специальное значение (01).

0	00..00	0000. 000	
79	78 64	63	0

1	00..00	0000. 000	
79	78 64	63	0

Рис. 17.13. Представление нуля в регистре стека сопроцессора

Значение нуля может быть сформировано в результате возникновения ситуации антипереполнения (см. далее), а также при работе команд с нулевыми операндами.

Бесконечность

Сопроцессор имеет средства в виде специальных битовых значений для представления бесконечности. Формат регистра стека сопроцессора, содержащего значение бесконечности, приведен на рис. 17.14.

0	11..11	10000 000	
79	78 64	63	0

1	11..11	10000 000	
79	78 64	63	0

Рис. 17.14. Представление значения бесконечности в регистре стека сопроцессора

Из рисунка видно, что значение бесконечности может иметь знак, при этом значения мантиссы и характеристики фиксированы. Именно в этом отличие значения бесконечности от остальных специальных значений.

Среди причин, приводящих к формированию значения бесконечности, можно выделить переполнение и деление на нуль. При формировании значения бесконечности в некотором регистре стека в соответствующем теге регистра TWR формируется специальное значение (10).

Нечисла

К *нечислам* относятся такие битовые последовательности в регистре стека сопроцессора, которые не совпадают ни с одним из рассмотренных ранее форматов значений. Нечисло должно иметь единичную мантиссу и любую характеристику, кроме 100...00, которая зарезервирована для значения бесконечность. Различают два типа нечисел:

- SNAN (Signaling Non a Number) — сигнальные нечисла;
- QNAN (Quiet Non A Number) — спокойные (тихие) нечисла.

Сигнальное нечисло — битовое значение с единичным значением полей характеристики и мантиссой, первый бит которой, следующий за первым единичным значащим битом, равен нулю (рис. 17.15, а). Сопроцессор реагирует на появление этого числа в регистре стека возбуждением исключения недействительной операции. Программисты могут формировать эти числа в регистре стека сопроцессора преднамеренно, например, чтобы искусственно возбудить в нужной ситуации указанное исключение. Очевидно, что именно по этой причине данные числа называются сигнальными. Если снять маску у флага недействительной операции в регистре CWR, то будет вызван обработчик, который выполнит заданные программистом действия.

а	x	11..11	10xxx	... xxx
	79	78 64	63	0
б	x	11..11	11xxx	... xxx
	79	78 64	63	0
в	1	11..11	11000	... 000
	79	78 64	63	0

Рис. 17.15. Представление нечисел в регистре стека сопроцессора

Спокойное нечисло — битовое значение с единичным значением полей характеристики и мантиссой, первые два бита которой равны единице (рис. 17.15, *б).*

Сопроцессор самостоятельно не формирует сигнальных чисел, но в качестве реакции на определенные исключения он может формировать спокойные нечисла, например нечисло *вещественной неопределенности* (рис. 17.15, б). Вещественная неопределенность формируется как маскированная реакция сопроцессора на исключение недействительной операции (см. раздел «Исключения сопроцессора и их обработка»). Другие спокойные нечисла могут формироваться после выполнения команд, в которых хотя бы один из операндов был спокойным нечислом. Это может породить «цепную реакцию», ведущую к ошибочному результату. Поэтому в процессе вычислений рекомендуется периодически контролировать результаты исполнения команд на предмет появления спокойных нечисел.

При формировании нечисла в некотором регистре стека в соответствующем теге регистра TWR формируется специальное значение (10).

Неподдерживаемые форматы

Необходимо иметь в виду, что помимо рассмотренных существует довольно много битовых наборов, которые можно представить в расширенном формате вещественного числа. Для большинства их значений формируется исключение недействительной операции.

Система команд сопроцессора

Система команд сопроцессора включает около 80 машинных команд. Рассмотрим их классификацию (рис. 17.16).

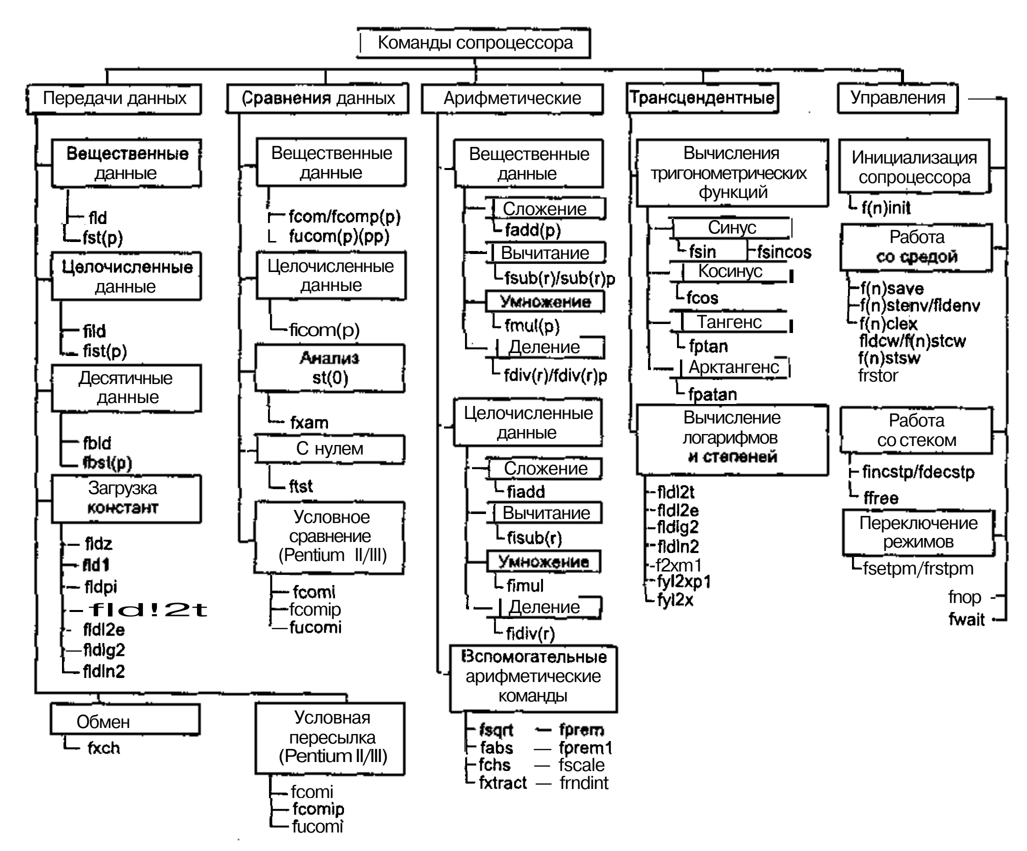


Рис. 17.16. Функциональная классификация команд сопроцессора

Мнемоническое обозначение команд сопроцессора характеризует особенности их работы и в связи с этим может представлять определенный интерес. Цоэтому коротко рассмотрим основные моменты образования названий команд.

- Все мнемонические обозначения начинаются с символа F (Float).
- Вторая буква мнемонического обозначения определяет тип операнда в памяти, с которым работает команда:
  - I — целое двоичное число;
  - B — целое десятичное число;
  - отсутствие буквы — вещественное число.
- Последняя буква P в мнемоническом обозначении команды означает, что последним действием команды обязательно является извлечение операнда из стека.
- Последняя или предпоследняя буква R (reversed) в мнемоническом обозначении команды означает реверсивное следование операндов при выполнении команд вычитания и деления, так как для них важен порядок следования операндов.

Полезной может оказаться и информация о машинных форматах команд сопроцессора. Если давать им общую характеристику, то важно отметить, что в целом система команд сопроцессора отличается большой гибкостью в выборе вариантов задания команд, реализующих определенную операцию, и их операндов. Минимальная длина команды сопроцессора — 2 байта. При обсуждении команд в тексте данной главы будет приводиться лишь их схема. Детально машинное представление команд сопроцессора и сами команды описаны в приложении.

Методика написания программ для сопроцессора имеет свои особенности. Главная причина здесь — в стековой организации сопроцессора. Для того чтобы написать программу для вычисления некоторого выражения, его необходимо предварительно преобразовать в удобный для программирования сопроцессора вид. Процесс преобразования напоминает подготовку выражения для метода трансляции, основанного на использовании *обратной польской записи* (ПОЛИЗ). Рассмотрим суть ПОЛИЗ на примере вычисления выражения

$$a + b \cdot c - d / (a + b).$$

Графически это выражение представляется в виде дерева (рис. 17.17).

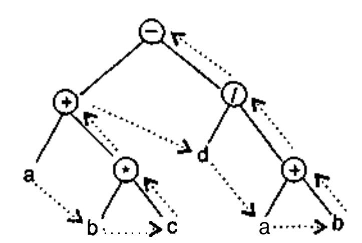


Рис. 17.17. Представление выражения в виде дерева

Листья ветвей дерева соответствуют операндам, а узлы — операциям. Пусть началом обхода будет лист самой левой ветви дерева. Тогда для получения обратной польской записи выражения необходимо двигаться по дереву слева направо, при этом узлы должны просматриваться только после обхода всех исходящих из него ветвей. В результате выражение будет выглядеть так:

abc x + dab + / -

Другое название такой формы записи — *постфиксная запись*. Это означает, что знак операции записывается после операндов, участвующих в операции. Постфиксная запись позволяет вычислять выражения за один проход с учетом приоритета арифметических операций.

Алгоритм вычисления выражений в постфиксной записи имеет следующий вид.

1. Выбрать очередной символ записи выражения в форме ПОЛИЗ.
2. Если очередной выбранный символ — операнд, то поместить его в стек, после чего вернуться к шагу 1.
3. Если очередной выбранный символ — знак операции, то выполнить ее с одним или двумя операндами на вершине стека. Результат операции необходимо поместить обратно на вершину стека.
4. Если в исходной записи выражения в форме ПОЛИЗ еще остались символы, то вернуться к шагу 1, иначе — на вершине стека получить результат вычисления выражения.

При разработке программ необходимо учитывать следующие факторы:

- ограниченность глубины стека сопроцессора;
- несовпадение форматов операндов;
- отсутствие поддержки на уровне команд сопроцессора некоторых операций, таких как возведение в степень, вычисление тригонометрических функций.

Для решения этих проблем вам придется разрабатывать дополнительный программный код, отклоняясь от классической (последовательной) обработки арифметических выражений, представленных в форме ПОЛИЗ. Однако стержнем программы останется строка в форме ПОЛИЗ.

Среди файлов к данной главе находятся файлы программы, с помощью которой вы можете облегчить себе задачу преобразования простого арифметического выражения в форму ПОЛИЗ. Она поддерживает ограниченное количество операций, но их набор вы можете расширить, дополнив конфигурационный файл polis.cfg и исходный текст программы.

Здесь возникает интересная алгоритмическая задача. В чем ее суть? Для тех операций, которые реализованы на уровне команд сопроцессора, особых проблем нет. И наоборот, для выражений, содержащих операции, не реализованные на уровне команд сопроцессора, возникает проблема преобразования исходной строки в форму ПОЛИЗ. В качестве примера можно привести такую распространенную операцию, как возведение в степень. В соответствующей программе, которую мы рассмотрим чуть позже, реализуется ряд операций, необходимых для возведения в степень. То есть в выражении, преобразованном в форму ПОЛИЗ, будут записаны некоторые операции, отсутствующие в исходной записи этого выражения. Это означает, что при формировании строки ПОЛИЗ программа должна выполнять неявную подстановку дополнительных операндов и операций в соответствии с некоторой формулой приведения.

Команды передачи данных

Группа команд передачи данных предназначена для организации обмена между регистрами стека, вершиной стека сопроцессора и ячейками оперативной памяти.

Команды этой группы имеют такое же значение для программирования сопроцессора, как команда MOV — для программирования основного процессора. С помощью команд передачи данных осуществляются все перемещения значений операндов в сопроцессор и из него. По этой причине для каждого из трех типов данных, с которыми может работать сопроцессор, существует своя подгруппа команд передачи данных. Собственно, на этом уровне все его умения по работе с различными форматами данных и заканчиваются. Главной функцией всех команд загрузки данных в сопроцессор является преобразование данных к единому представлению в виде вещественного числа расширенного формата. Это же касается и обратной операции — сохранения в памяти данных из сопроцессора.

Команды передачи данных можно разделить на следующие группы:

- команды передачи данных в вещественном формате;
- команды передачи данных в целочисленном формате;
- команды передачи данных в десятичном формате.

Далее перечислены команды передачи данных в вещественном формате.

- FLD источник — загрузка вещественного числа из области памяти на вершину стека сопроцессора.
- FST приемник — сохранение вещественного числа из вершины стека сопроцессора в память. Как следует из анализа мнемокода команды (отсутствует символ P), сохранение числа в памяти не сопровождается выталкиванием его из стека, то есть текущая вершина стека сопроцессора не меняется (поле TOP не меняется).
- FSTP приемник — сохранение вещественного числа из вершины стека сопроцессора в память. В отличие от предыдущей команды, в конце мнемонического обозначения данной команды присутствует символ P, что означает выталкивание вещественного числа из стека после его сохранения в памяти. Команда изменяет поле TOP, увеличивая его на единицу. Вследствие этого вершиной стека становится следующий больший по своему физическому номеру регистр стека сопроцессора.

Далее перечислены команды передачи данных в целочисленном формате.

- FILD источник — загрузка целого числа из памяти на вершину стека сопроцессора.
- FIST приемник — сохранение целого числа из вершины стека сопроцессора в память. Сохранение целого числа в памяти не сопровождается выталкиванием его из стека, то есть текущая вершина стека сопроцессора не изменяется.
- FISTP приемник — сохранение целого числа из вершины стека в память. Аналогично сказанному ранее о команде FSTP, последним действием команды является выталкивание числа из стека с одновременным преобразованием его в целое значение.

И наконец, осталось перечислить команды передачи данных в десятичном формате.

- FBLD источник — загрузка десятичного числа из памяти на вершину стека сопроцессора.

- FBSTP приемник — сохранение десятичного числа из вершины стека сопроцессора в области памяти. Значение выталкивается из стека после преобразования его в формат десятичного числа. Заметьте, что для десятичных чисел нет команды сохранения значения в памяти без выталкивания из стека.

К группе команд передачи данных можно отнести также команду обмена вершины регистрового стека ST(0) с любым другим регистром стека сопроцессора ST(i):

```
fxch   st(i)
```

Действие команд загрузки FLD, FILD и FBLD можно сравнить с командой PUSH основного процессора. Аналогично ей (PUSH уменьшает значение в регистре SP) команды загрузки сопроцессора перед сохранением значения в регистровом стеке сопроцессора вычитают из содержимого поля TOP регистра состояния SWR единицу. Это означает, что вершиной стека становится регистр с физическим номером на единицу меньше. При этом возможно переполнение стека. Так как стек сопроцессора состоит из ограниченного числа регистров, то в него может быть записано максимум восемь значений. Из-за кольцевой организации стека девятое записываемое значение затирает первое. Программа должна иметь возможность обработать такую ситуацию. По этой причине почти все команды, помещающие свой операнд в стек сопроцессора, после уменьшения значения поля TOP проверяет регистр — кандидат на новую вершину стека — на предмет его занятости. Для анализа этой и подобных ситуаций используется регистр TWR, содержащий слово тегов (см. рис. 17.1). Наличие регистра тегов в архитектуре сопроцессора позволяет освободить программиста от разработки сложной процедуры распознавания содержимого регистров сопроцессора и дает самому сопроцессору возможность фиксировать определенные ситуации, например попытку чтения из пустого регистра или запись в непустой регистр. Возникновение таких ситуаций фиксируется в регистре состояния SWR (см. рис. 17.2), предназначенном для сохранения общей информации о сопроцессоре. Используя специальные команды сопроцессора, можно извлечь из него или, напротив, записать в него информацию (см. далее подраздел «Команды управления сопроцессором» данного раздела).

В качестве примера определим несколько констант и выполним их пересылки между оперативной памятью и регистрами сопроцессора, а также между регистрами стека. При этом будем следить за состоянием стека сопроцессора. В ходе реализации этих операций вполне закономерно возникает вопрос о том, имеются ли какие-либо средства для наблюдения за состоянием регистров сопроцессора, аналогично тому, как это делалось при работе с основным процессором. Для него, как вы помните, мы использовали отладчик Turbo Debuger. Оказывается, что далеко ходить не нужно. Тот же Turbo Debuger предоставляет нам эту возможность. Чтобы не нарушать структуру общего изложения, материал по Turbo Debuger вынесен в отдельный раздел «Использование отладчика». С ним можно ознакомиться прямо сейчас, после чего вернуться обратно. При наличии у вас другого отладчика, отличного от Turbo Debuger, следует обратиться к его описанию и поискать аналогичные средства.

Введите программу из листинга 17.1. Выполните ее под управлением отладчика и проследите за тем, как меняется состояние регистров сопроцессора в окне Numeric processor.

Листинг 17.1. Исследование команд передачи данных

```
;-------fpu.asm-------------------------------
.586p
masm
model   use16 small
.stack  100h
.data                       ;сегмент данных
ch_dt   dt  43567           ;ch_dt=00 00 00 00 00 00 00 04 35 67
x   dw  3                   ;x=00 03
y_real  dq  34e7            ;y_real=41 b4 43 fd 00 00 00 00
ch_dt_st    dt  0
x_st    dw  0
y_real_st   dq  0
.code
main    proc                ;начало процедуры main
    mov     ax, @data
    mov     ds, ax
    fbld    ch_dt           ;st(0)=43567
    fild    x               ;st(1)=43567, st(0)=3
    fld     y_real          ;st(2)=43567, st(1)=3, st(0)=340000000
    fxch     st(2)          ;st(2)=340000000, st(1)=3 , st(0)=43567
    fbstp    ch_dt_st       ;st(1)=340000000,  st(0)=3
                            ;ch_dt_st=00 00 00 00 00 00 00 04 35 67
    fistp    x_st           ;st(0)=340000000,  x_st=00 03
    fstp     y_real_st      ;y_real_st=41 b4 43 fd 00 00 00 00
exit:
    mov ax, 4c00h
    int 21h
main    endp
end  main
```

Исследовать работу этой программы полезно с открытым окном Dump, так как при этом хорошо видны различия в представлении типов данных.

Команды загрузки констант

Основным назначением сопроцессора является поддержка вычислений с плавающей точкой. В математических вычислениях довольно часто встречаются предопределенные константы, и сопроцессор хранит значения некоторых из них. Другая причина использования этих констант заключается в том, что для определения их в памяти (в расширенном формате) требуется 10 байт, а это для хранения, например, единицы расточительно (сама команда загрузки константы, хранящейся в сопроцессоре, занимает два байта). В формате, отличном от расширенного, эти константы хранить не имеет смысла, так как теряется время на их преобразование в тот же расширенный формат. Для каждой предопределенной константы существует специальная команда, которая производит загрузку ее на вершину регистрового стека сопроцессора:

- FLDZ — загрузка нуля;
- FLD1 — загрузка единицы;
- FLDPI — загрузка числа π;
- FLDL2T — загрузка двоичного логарифма десяти;
- FLDL2E — загрузка двоичного логарифма экспоненты (числа e);
- FLDLG2 — загрузка десятичного логарифма двойки;
- FLDLN2 — загрузка натурального логарифма двойки.

Команды сравнения данных

Команды сравнения данных сравнивают значение числа в вершине стека и операнда, указанного в команде.

- FCOM [операнд_в_памяти] — команда без операндов сравнивает два значения: одно находится в регистре ST(0), другое в регистре ST(1). Если указан операнд [операнд_в_памяти], то сравнивается значение в регистре ST(0) стека сопроцессора со значением в памяти.
- FCOMP операнд — команда сравнивает значение в вершине стека сопроцессора ST(0) со значением операнда, который находится в регистре или в памяти. Последним действием команды является выталкивание значения из ST(0).
- FCOMPP операнд — команда аналогична по действию команде FCOM без операндов, но последним ее действием является выталкивание из стека значений обоих регистров, ST(0) и ST(1).
- FICOM операнд_в_памяти — команда сравнивает значение в вершине стека сопроцессора ST(0) с целым операндом в памяти. Длина целого операнда — 16 или 32 бита, то есть это целое слово и короткое целое (см. табл. 17.1).
- FICOMP операнд — команда сравнивает значение в вершине стека сопроцессора ST(0) с целым операндом в памяти. После сравнения и установки битов C3...C0 команда выталкивает значение из ST(0). Длина целого операнда — 16 или 32 бита, то есть это целое слово и короткое целое (см. табл. 17.1).
- FTST — команда не имеет операндов и сравнивает значения в ST(0) с нулем (значением 00).

Предыдущие команды сравнения работают корректно, если операнды в них являются целыми или вещественными числами. Когда один из операндов оказывается нечислом, то фиксируется исключение недействительной ситуации, а коды условия C3...C0 соответствуют исключительной ситуации несравнимых или неупорядоченных операндов. Само же действие сравнения не производится. Процессор предоставляет три команды, позволяющие все же произвести сравнение таких операндов, но как вещественных чисел без учета их порядков.

- FUCOM st(i) — команда сравнивает значения (без учета их порядков) в регистрах стека сопроцессора ST(0) и ST(i).
- FUCOMP st(i) — команда сравнивает значения (без учета их порядков) в регистрах стека сопроцессора ST(0) и ST(i). Последним действием команды является выталкивание значения из вершины стека.
- FUCOMPP st(i) — команда сравнивает значения (без учета их порядков) в регистрах стека сопроцессора ST(0) и ST(i). Последние два действия команды одинаковы — выталкивание значения из вершины стека.

В результате работы команд сравнения в регистре состояния устанавливаются следующие значения битов кода условия C3, C2, C0:

- если ST(0) > операнда, то 000;
- если ST(0) < операнда, то 001;
- если ST(0) = операнду, то 100;
- если операнды неупорядочены, то 111.

Для того Чтобы получить возможность реагировать на эти коды командами условного перехода основного процессора (вспомните, что они реагируют на флаги в EFLAGS), нужно как-то записать сформированные биты условия C3, C2, C0 в регистр EFLAGS. В системе команд сопроцессора существует команда **FSTSW**, которая позволяет запомнить слово состояния сопроцессора в регистре AX или ячейке памяти. Далее значения нужных битов извлекаются и анализируются командами основного процессора. Например, запись старшего байта слова состояния, в котором находятся биты C0...C3, в младший байт регистра EFLAGS/FLAGS осуществляется командой **SAHF**. Эта команда записывает содержимое AH в младший байт регистра EFLAGS/FLAGS. После этого бит C0 записывается на место флага CF, C2 — на место PF, C3 — на место ZF. Бит C1 выпадает из общего правила, так как в регистре флагов на месте соответствующего ему бита находится единица. Анализ этого бита нужно проводить с помощью логических команд основного процессора. Зная все это, вам остается, исходя из особенностей своего алгоритма, применять те команды условного перехода, которые анализируют состояние указанных флагов.

В качестве иллюстрации рассмотрим программу разбиения массива вещественных чисел в формате двойного слова на два массива. В первый массив поместим все элементы, которые больше или равны нулю, а во второй — меньше нуля (листинг 17.2).

Листинг 17.2. Исследование команд сравнения данных

```
.586p
masm
model   use16 small
.stack  100h
.data   ;сегмент данных
;исходный массив
mas dd  -2.0, 45.7, -9.4, 7.3, 60.3, -58.44, 890e7, -98746e3
mas_h_0 dd  8 dup (0)      ;массив значений, больше либо равных 0,
i_mas_h_0   dd  0          ;текущий индекс в mas_h_0
mas_l_0 dd  8 dup (0)      ;массив значений меньших 0
i_mas_l_0   dd  0          ;текущий индекс в mas_l_0
.code
main    proc               ;начало процедуры main
    mov ax, @data
    mov ds, ax
    xor esi, esi
    mov cx, 8              ;счетчик циклов
    finit                  ;приведение сопроцессора в начальное состояние
    fldz                   ;загрузка нуля в st(0)
cycl:
    fcom mas[esi*4]        ;сравнение нуля в st(0)
                           ;с очередным элементом массива mas
    fstsw   ax             ;сохранение swr в регистре ax
    sahf                   ;запись swr->ax-> регистр флагов
    jp  error              ;переход по "плохому" операнду в команде fcom
    jc  hi_0;переход, если mas[esi*4]>= 0 (mas[esi*4]>=st(0))
;пересылка операнда mas[esi*4], меньшего 0. в массив mas_l_0
    mov eax, mas[esi*4]
    mov edi, i_mas_l_0
    mov mas_l_0[edi*4], eax
    inc i_mas_l_0
    jmp cycl_bst
hi_0:
;пересылка операнда mas[esi*4], большего или равного 0, в массив mas_h_0
    mov eax, mas [esi*4]
```

продолжение ➩

Листинг 17.2 *(продолжение)*

```
    mov edi, i_mas_h_0
    mov raas_h_0[edi*4], eax
    inc i_mas_h_0
cycl_bst:
    inc si
    loop cycl
error:
;здесь можно вывести сообщение об ошибке в задании операндов
exit:
    mov ax, 4c00h
    int 21h
main    endp
end  main
```

К группе команд сравнения данных логично отнести и команду FXAM, которая анализирует операнд в вершине стека сопроцессора ST(0) и формирует значение битов C0, C1, C2, C3 в регистре состояния сопроцессора SWR. По состоянию этих битов можно судить:

* о знаке мантиссы — знаковый бит операнда в ST(0) заносится в бит C0 регистра SWR;
* корректности записи вещественного числа в ST(0) — идентифицируются пустой регистр, корректное вещественное число, нечисло и неизвестный формат;
* типе специального численного значения: бесконечность, нуль, денормализованный операнд.

Подробное описание команды FXAM можно найти в приложении.

Арифметические команды

Команды сопроцессора, входящие в группу арифметических команд, реализуют четыре основные арифметические операции — сложение, вычитание, умножение и деление. Имеется также несколько дополнительных команд, предназначенных для повышения эффективности использования основных арифметических команд. С точки зрения типов операндов арифметические команды сопроцессора можно разделить на команды, работающие с вещественными и целыми числами. Рассмотрим их.

Целочисленные арифметические команды

Целочисленные арифметические команды предназначены для работы на тех участках вычислительных алгоритмов, где в качестве исходных данных используются целые числа в памяти в формате слово и короткое слово, имеющие размерность 16 и 32 бита.

При рассмотрении целочисленных арифметических команд обратите внимание на большую гибкость задания операндов в этих командах.

* FIADD источник — команда складывает значения ST(0) и целочисленного источника, в качестве которого выступает 16- или 32-разрядный операнд в памяти. Результат сложения запоминается в регистре стека сопроцессора ST(0).
* FISUB источник — команда вычитает значение целочисленного источника из ST(0). Результат вычитания запоминается в регистре стека сопроцессора ST(0). В качестве источника выступает 16- или 32-разрядный целочисленный операнд в памяти.

- FIMUL источник — команда умножает значение целочисленного источника на содержимое ST(0). Результат умножения запоминается в регистре стека сопроцессора ST(0). В качестве источника выступает 16- или 32-разрядный целочисленный операнд в памяти.
- FIDIV источник — команда делит содержимое ST(0) на значение целочисленного источника. Результат деления запоминается в регистре стека сопроцессора ST(0). В качестве источника выступает 16- или 32-разрядный целочисленный операнд в памяти.

Для команд, реализующих арифметические действия деления и вычитания, важен порядок расположения операндов. По этой причине система команд сопроцессора содержит соответствующие реверсивные команды, повышающие удобство программирования вычислительных алгоритмов. Чтобы отличить эти команды от обычных команд деления и вычитания, их мнемокоды оканчиваются символом R.

- FISUBR источник — команда вычитает значение ST(0) из целочисленного источника. Результат вычитания запоминается в регистре стека сопроцессора ST(0). В качестве источника выступает 16- или 32-разрядный целочисленный операнд в памяти.
- FIDIVR источник — команда делит значение целочисленного источника на содержимое ST(0). Результат деления запоминается в регистре стека сопроцессора ST(0). В качестве источника выступает 16- или 32-разрядный целочисленный операнд в памяти.

Рассмотрим пример программы вычисления значения u (листинг 17.3):

$$u = \frac{x - u}{a}, \text{ если } a \neq 0;$$

$$u = x + y, \text{ если } a = 0.$$

Все переменные x, y и a целого типа в формате слова. Результат необходимо сохранить в ячейке памяти в формате десятичного числа.

Листинг 17.3. Исследование целочисленных арифметических команд

```
.586p
masm
model   use16 small
 .stack  100h
 .data   ;сегмент данных
; исходный массив
a   dw  0
x   dw  8
y   dw  4
u   dt  0
 .code
main    proc
    mov ax, @data
    mov ds, ax
    finit       ;приведение сопроцессора в начальное состояние
    fild a       ;загрузка значение a в st(0)
    fxam        ;определяем тип a
    fstsw  ax  ;сохранение swr в регистре ax sahf
    ;запись swr->ax-> регистр флагов
```

продолжение ⇒

Листинг 17.3 *(продолжение)*

```
    jc   no_null
    jp   no_null
    jnz  no_null
;вычисление формулы u=x+y:
    fild    x
    fiadd   y
    fbstp   u
    jmp     exit
no_null:
;вычисление формулы u=(x-y)/a:
    fild    x
    fisub   y
    fidiv   a
    fbstp   u
exit:
    mov     ax, 4c00h
    int     21h
main    endp
end main
```

Вещественные арифметические команды

Схема расположения операндов вещественных команд традиционна для команд сопроцессора. Один из операндов располагается в вершине стека сопроцессора — регистре ST(0), куда после выполнения команды записывается и результат, а второй операнд может быть расположен либо в памяти, либо в другом регистре стека сопроцессора. Допустимыми типами операндов в памяти являются все перечисленные ранее вещественные форматы за исключением расширенного.

В отличие от целочисленных арифметических команд, вещественные арифметические команды допускают большее разнообразие в сочетании местоположения операндов и самих команд для выполнения конкретного арифметического действия. Так, например, можно выделить три возможных варианта команды сложения. В дополнение к этим трем вариантам существует еще одна команда сложения, производящая дополнительное действие — удаление значения из стека.

- FADD — команда складывает значения в ST(0) и ST(1). Результат сложения запоминается в регистре стека сопроцессора ST(0).
- FADD источник — команда складывает значения ST(0) и источника, представляющего адрес ячейки памяти. Результат сложения запоминается в регистре стека сопроцессора ST(0).
- FADD st(i),st — команда складывает значение в регистре стека сопроцессора ST(i) со значением в вершине стека ST(0). Результат сложения запоминается в регистре ST(i).
- FADDP st(i),st — команда производит сложение вещественных операндов аналогично команде FADD st(i),st, однако последним действием команды является выталкивание значения из вершины стека сопроцессора ST(0). Результат сложения остается в регистре ST(i-1).

Для выполнения операции вычитания также имеется большой набор команд.

- FSUB — команда вычитает значение в ST(1) из значения в ST(0). Результат вычитания запоминается в регистре стека сопроцессора ST(0).

- FSUB источник — команда вычитает значение источника из значения в ST(0). Источник представляет адрес ячейки памяти, содержащей допустимое вещественное число. Результат сложения запоминается в регистре стека сопроцессора ST(0).
- FSUB st(i),st — команда вычитает значение в вершине стека ST(0) из значения в регистре стека сопроцессора ST(i). Результат вычитания запоминается в регистре стека сопроцессора ST(i).
- FSUBP st(i),st — команда вычитает вещественные операнды аналогично команде FSUB st(i),st. Последним действием команды является выталкивание значения из вершины стека сопроцессора ST(0). Результат вычитания остается в регистре ST(i–1).

Для удобства группа команд вычитания вещественных чисел дополнена командами реверсивного вычитания.

- FSUBR st(i),st — команда вычитает значение в вершине стека ST(0) из значения в регистре стека сопроцессора ST(i). Результат вычитания запоминается в вершине стека сопроцессора — регистре ST(0).
- FSUBRP st(i),st — команда производит вычитание подобно команде FSUBR st(i),st. Последним действием команды является выталкивание значения из вершины стека сопроцессора ST(0). Результат вычитания остается в регистре ST(i–1).

Изучая команды умножения вещественных операндов, обратите внимание не то, что операнды располагаются исключительно в стеке сопроцессора.

- FMUL — команда не имеет операндов. Умножает значения в ST(0) на содержимое в ST(1). Результат умножения запоминается в регистре стека сопроцессора ST(0).
- FMUL st(i) — команда умножает значение в ST(0) на содержимое регистра стека ST(i). Результат умножения запоминается в регистре стека сопроцессора ST(0).
- FMUL st(i),st — команда умножает значения в ST(0) на содержимое произвольного регистра стека ST(i). Результат умножения запоминается в регистре стека сопроцессора ST(i).
- FMULP st(i),st — команда производит умножение подобно команде FMUL st(i),st. Последним действием команды является выталкивание значения из вершины стека сопроцессора ST(0). Результат умножения остается в регистре ST(i–1).

И наконец, рассмотрим команды, реализующие деление вещественных данных. Подобно командам умножения, операнды этих команд располагаются в стеке сопроцессора:

- FDIV — команда (без операндов) делит содержимого регистра ST(0) на значение регистра сопроцессора ST(1). Результат деления запоминается в регистре стека сопроцессора ST(0).
- FDIV st(i) — команда делит содержимое регистра ST(0) на содержимое регистра сопроцессора ST(i). Результат деления запоминается в регистре стека сопроцессора st(0).
- FDIV st(i),st — команда производит деление аналогично команде FDIV st(i), но результат деления запоминается в регистре стека сопроцессора ST(i).

- FDIVP st(i),st — команда производит деление аналогично команде FDIV st(i),st. Последним действием команды является выталкивание значения из вершины стека сопроцессора ST(0). Результат деления остается в регистре ST(i-1).

Для реализации деления в сопроцессоре также предусмотрены две реверсивные команды, отличительным признаком которых является наличие символа R в качестве последнего или предпоследнего символа мнемокода:

- FDIVR st(i),st — команда делит содержимое регистра ST(i) на содержимое вершины регистра сопроцессора ST(0). Результат деления запоминается в регистре стека сопроцессора ST(0).
- FDIVRP st(i),st — команда делит содержимое регистра ST(i) на содержимое вершины регистра сопроцессора ST(0). Результат деления запоминается в регистре стека сопроцессора ST(i), после чего производится выталкивание содержимого ST(0) из стека. Результат деления остается в регистре ST(i-1).

Разработаем программу вычисления факториала числа 10 (листинг 17.4):

$$y=\prod_{i=1}^{10} i .$$

Напомним, что вычисление факториала заключается в выполнении последовательного умножения 1 • 2 • 3 · ...• (г—1) • г.

Листинг 17.4. Исследование вещественных арифметических команд

```
.586p
masm
model   use16 small
.stack  100h
.data                       ;сегмент данных
;исходный массив
i   equ 10
y   dq  0
.code
main    proc
    mov     ax, @data
    mov     ds, ax
    finit                   ;приведение сопроцессора в начальное состояние
    fldl                    ;st(0)=l!
    fldl                    ;st(0)=i=l. st(1)=l!
    fst     y
    mov     cx, i-1         ;первый элемент уже вычислили
cycl:
    fldl
    fadd    p
    fmul    st(1), st(0);st(0)=i=2, 3.., st(1)=i!
    fldl
    fdiv    st(0),, st(2);1/i!
    fadd    y               ;накопление суммы
    fstp    y               ;сохранение суммы
    loop    cycl
exit:
    mov     ax, 4c00h
    int     21h
main    endp
end main
```

Дополнительные арифметические команды

Далее перечислены команды, входящие в группу дополнительных арифметических команд/

- FSQRT — вычисление квадратного корня из значения, находящегося в вершине стека сопроцессора — регистре ST(0). Команда не имеет операндов. Результат вычисления помещается в регистр ST(0). Следует отметить, что данная команда обладает определенными достоинствами. Во-первых, результат извлечения достаточно точен, во-вторых, скорость исполнения чуть больше скорости выполнения команды деления вещественных чисел FDIV;
- FABS — вычисление модуля значения, находящегося в вершине стека сопроцессора — регистре ST(0). Команда не имеет операндов. Результат вычисления помещается в регистр ST(0);
- FCHS — изменение знака значения, находящегося в вершине стека сопроцессора — регистре ST(0). Команда не имеет операндов. Результат вычисления помещается обратно в регистр ST(0). Отличие команды FCHS от команды FABS в том, что команда FCHS только инвертирует знаковый разряд значения в регистре ST(0), не меняя значения остальных битов. Команда вычисления модуля FABS при наличии отрицательного значения в регистре ST(0), наряду с инвертированием знакового разряда, выполняет изменение остальных битов значения таким образом, чтобы в ST(0) получилось соответствующее положительное число.
- FXTRACT — команда выделения порядка и мантиссы значения, находящегося в вершине стека сопроцессора — регистре ST(0). Команда не имеет операндов. Результат выделения помещается в два регистра стека — мантисса в ST(0), а порядок в ST(1). При этом мантисса представляется вещественным числом с тем же знаком, что и у исходного числа, и порядком равным нулю. Порядок, помещенный в ST(1), представляется как истинный порядок, то есть без константы смещения, в виде вещественного числа со знаком и соответствует величине Р формулы (17.1).

На последнюю команду следует обратить особое внимание, так как она позволяет выделить и затем по отдельности проанализировать порядок и мантиссу числа, содержащегося в вершине стека. Для того чтобы разобраться с тонкостями работы этой команды, изучите листинг 17.5. Исходное десятичное число для этой программы равно 45,56. Оно взято из рассмотренного ранее примера по определению десятичного числа в расширенном формате (см. раздел «Форматы данных»).

Листинг 17.5. Исследование работы команды FXTRACT

```
.586p
masm
model   use16 small
 .stack  100h
 .data                  ;сегмент данных
m_dt    dt  0
ch_dt   dt  0
y_real  dt  45.56       ;y_real=4004 b63d 70a3 d70a 3d71
 .code
main    proc            ;начало процедуры main
    mov ax, @data
```

продолжение ↗

Листинг 17.5 *(продолжение)*

```
    mov ds, ax
    fld y_real   ;st(0)=4004 b63d 70a3 d70a 3d71 (45.56)
    fxtract      ;st(1)=4001 a000 ... 0000 (5),
                 ;st(0)=3fff b63d 70a3 d70a 3d71 (1.42375)
    fstp m_dt    ;st(0)=4001 a000 ... 0000
                 ;m_dt=3fff b63d 70a3 d70a 3d71 (1.42375)
    fstp ch_dt   ;m_dt=3fff b63d 70a3 d70a 3d71 (1.42375)
                 ;ch_dt=4001 a000 ... 0000 (5)
exit:
    mov ax, 4c00h
    int 21h
main    endp
end  main
```

Введите эту программу, получите исполняемый модуль и загрузите его в отладчик Turbo Debuger. Раскройте на весь экран окно Numeric processor, чтобы наблюдать за содержимым регистров стека сопроцессора в шестнадцатеричном формате. В пошаговом режиме проследите за теми действиями, которые выполняет команда FXTRACT. Вы обнаружите, что команда FXTRACT выделяет характеристику вещественного числа в расширенном формате по формуле (17.3) вычисляет истинный порядок числа, представляет его как число со знаком и заносит в регистр ST(1). Не забывайте, что это порядок нормализованного числа в двоичном формате, поэтому для нашего примера он равен 5 (в формуле (17.2) N = 2). Далее команда FXTRACT формирует нулевой порядок значения в ST(0), занося в поле характеристики значение 3fffh(16383_{10}). На этом ее работа заканчивается. Следующие за FXTRACT команды сохраняют значения мантиссы и порядка в памяти как самостоятельные значения. Восстановить исходное значение можно командой масштабирования FSCALE, которая к тому же позволяет легко производить умножение на целочисленную степень числа 2. Эта команда изменяет порядок значения, находящегося в вершине стека сопроцессора — регистре ST(0), на величину в ST(1). Команда не имеет операндов. Величина в ST(1) рассматривается как число со знаком. Его прибавление к полю порядка вещественного числа в ST(0) означает его умножение на величину $2^{st(1)}$,то есть ST(0) = ST(0) • $2^{st(1)}$.

С помощью данной команды удобно масштабировать на степень двойки некоторую последовательность чисел в памяти. Для этого достаточно поочередно загружать числа последовательности в вершину стека, после чего применять команду FSCALE и сохранять значения обратно в памяти. Команда FSCALE по алгоритму работы является обратной команде FXTRACT. Чтобы убедиться в этом, модифицируем листинг 17.5, превратив его в листинг 17.6.

Листинг 17.6. Исследование работы команды FSCALE

```
.586p
masm
model   use16 small
 .stack  100h
 .data                  ;сегмент данных
y_real  dt  45.56      ;y_real=4004 b63d 70a3 d70a 3d71
 .code
main    proc           ;начало процедуры main
    mov ax, @data
    mov ds, ax
    fld y_real         ;st(0)=4004 b63d 70a3 d70a 3d71 (45.56)
```

```
    fxtract     ;st(1)=4001 a000 ... 0000 (5)
                ;st(0)=3fff b63d 70a3 d70a 3d71 (1.42375)
    fscale      ;st(0)= 4004 b63d 70a3 d70a 3d71 (45.56)
exit:
...
```

Сопроцессор имеет программно-аппаратные средства для округления тех результатов работы команд, которые не могут быть точно представлены. Но операция округления может быть проведена к значению в регистре ST(0) и принудительно, для этого предназначена последняя команда в группе дополнительных команд — команда FRNDINT. Эта команда округляет до целого значение, находящееся в вершине стека сопроцессора — регистре ST(0). Команда не имеет операндов.

Возможны четыре режима округления величины в ST(0), которые определяются значениями в двухразрядном поле RC управляющего регистра сопроцессора. Как изменить режим округления? С помощью двух команд, FSTCWR и FLDCWR, которые, соответственно, записывают в память содержимое управляющего регистра сопроцессора и восстанавливают его обратно. Таким образом, пока содержимое этого регистра находится в памяти, вы имеете возможность установить необходимое значение поля RC.

Исследуем работу команды FRNDINT на следующем примере (листинг 17.7).

Листинг 17.7. Исследование работы команды FRNDINT

```
.586p
masm
model   use16 small
 .stack  100h
 .data                  ;сегмент данных
mem16   dw  0
y_real  dt   10.0
x_3 dd  3.0
 .code
main    proc            ;начало процедуры main
    mov     ax, @data
    mov     ds, ax
    fld     y_real
    fld     x_3
    fdiv                ;st(0)=3.333...33
    fstcw   mem16
    and     mem16, 1111001111111111b
    fldcw   mem16
    frndint             ;rc=0, st(0)=3
    fld y_real
    fld x_3
    fdiv                ;st(0)=3.333...33
    fstcw   mem16
    and     mem16, 1111001111111111b
    or      mem16, 1111011111111111b
    fldcw   mem16
    frndint             ;rc=01, st(0)=3
    fld     y_real
    fld     x_3
    fdiv                ;st(0)=3.333...33
    fstcw   mem16
    and     mem16, 1111001111111111b
    or      mem16, 1111101111111111b
    fldcw   mem16
    frndint             ;rc=10, st(0)=4
    fld     y_real
```

продолжение ⇨

Листинг 17.7 *(продолжение)*

```
    fld     x_3
    fdiv            ;st(0)=3.333...33
    fstcw   mem16
    and     mem16, 1111001111111111b
    or      mem16, 1111111111111111b
    fldcw   mem16
    frndint         :rc=II. st(0)=3
exit:
    mov     ax, 4c00h
    int     21h
main    endp
end     main
```

Рассмотрим еще один пример. Разработаем программу вычисления следующего выражения (листинг 17.8):

$$z=\left(\sqrt{|x|}-y\right)^2.$$

Результат сохраним в той же ячейке памяти, в которой хранится переменная *y*, в формате двойного слова.

Листинг 17.8. Вычисление выражения

```
.586p
masm
model  use16 small
.stack  100h
.data       ;сегмент данных
;исходные данные:
x   dd  -29e-4
y   dq  4.6
z   dd  0
.code
main    proc
    mov ax,@data
    mov ds,ax
    finit   ;приведение сопроцессора в начальное состояние
    fld x   ;st(0)=x
    fabs     ;st(0)=|x|
    fsqrt
    fsub y  ;st(0)=sqrt|x|-y
;квадрат придется вычислять через умножение
    fst st(1)
    fmul
    fst z
exit:
    mov ax,4c00h
    int 21h
main    endp
end     main
```

Как вы знаете, в сопроцессоре нет команды возведения в степень. Далее мы покажем, как можно решить эту проблему.

Команды трансцендентных функций

Сопроцессор имеет ряд команд, предназначенных для вычисления значений тригонометрических функций, таких как синус, косинус, тангенс, арктангенс, а также значений логарифмических и показательных функций. Наличие этих команд значи-

тельно облегчает жизнь программисту, вынужденному интенсивно заниматься разработкой вычислительных алгоритмов. Выигрыш налицо. Во-первых, отпадает необходимость самому разрабатывать соответствующие подпрограммы. Во-вторых, точность результатов выполнения трансцендентных команд очень высока.

Необходимо обратить внимание читателя на то, что значения аргументов в командах, вычисляющих результат тригонометрических функций, должны задаваться в радианах. В связи с этим приведем правило пересчета. Для нахождения радианной меры угла по его градусной мере необходимо число градусов умножить на я/180 (=0,017453), число минут - на $\pi/(180 \cdot 60)$ ($\approx$0,000291), а число секунд - на $\pi/(180 \cdot 60 \cdot 60)$ ($\approx$0,000005) и найденные произведения сложить.

Далее перечислены команды трансцендентных функций.

- FCOS — команда вычисляет косинус угла, находящийся в вершине стека сопроцессора — регистре ST(0). Команда не имеет операндов. Результат возвращается в регистр ST(0).
- FSIN — команда вычисляет синус угла, находящийся в вершине стека сопроцессора — регистре ST(0). Команда не имеет операндов. Результат возвращается в регистр ST(0).
- FSINCOS — команда вычисляет синус и косинус угла, находящиеся в вершине стека сопроцессора — регистре ST(0). Команда не имеет операндов. Результат возвращается в регистрах ST(0) и ST(1). При этом синус помещается в ST(0), а косинус — в ST(1).
- FPTAN — команда вычисляет *частичный* тангенс угла, находящийся в вершине стека сопроцессора — регистре ST(0). Команда не имеет операндов. Результат возвращается в регистрах ST(0) и ST(1).
- FPRATAN — команда вычисляет *частичный* арктангенс угла, находящийся в вершине стека сопроцессора — регистре ST(0). Команда не имеет операндов. Результат возвращается в регистрах ST(0) и ST(1).

Интересна история команды FPTAN. В отличие от команд вычисления синуса и косинуса, появившихся только в системе команд сопроцессора i387, команда FPTAN присутствовала еще *ц* системе команд сопроцессора i8087. Выполнение ее имело следующую особенность: результат команды возвращался в виде двух значений — в регистрах ST(0) и ST(1), но ни одно из них не являлось истинным значением тангенса. Истинное значение получается лишь после выполнения операции деления ST(0)/ST(1). Таким образом, для получения тангенса требуется еще команда деления. Зачем это нужно? Ранее мы упомянули о том, что команды для вычисления синуса и косинуса появились только в сопроцессоре i387, поэтому возникает вопрос о том, как вычислялись значения этих функций в ранних сопроцессорах. Чтобы понять это, введем необходимые обозначения: команда FPTAN вычисляет частичный тангенс числа z, значение которого находится в границах $0 < z <$ л/4. Результат работы команды FPTAN, как уже было отмечено, размещается в двух местах: x — в регистре ST(0), y — в регистре ST(1). Значения x, y и z таковы, что удовлетворяют соотношению

$$tg(z/2) = y/x.$$

Более того, используя значения x и y, можно получить значения остальных тригонометрических функций. Для этого используются следующие соотношения:

$$\mathrm{ctg}(z/2) = x/y;$$
$$\sin(z) = (2y/x)/((1 + (y/x)^2);$$
$$\cos(z) = (1 - y/x)^2)/(1 + (y/x)^2).$$

Теперь вам, наверное, понятен термин «частичный» тангенс. К счастью, в процессоре i387 появились самостоятельные команды для вычисления синуса и косинуса, вследствие чего отпала необходимость составлять соответствующие подпрограммы. Что же до команды FPTAN, то она по-прежнему выдает два значения в регистрах ST(0) и ST(1). Но теперь уже значение в ST(1) всегда равно единице, а это означает, что в ST(0) находится истинное значение тангенса числа, находившегося в ST(0) до выдачи команды FPTAN.

Рассмотрим теперь подробней команду вычисления частичного арктангенса угла FPATAN. Если использовать введенные нами при рассмотрении команды FPTAN обозначения, то для команды FPATAN действует следующее отношение:

$$z = \mathrm{arctg}(x/y).$$

Значения x и y размещаются в стеке следующим образом: x — в регистре ST(0), y — в регистре в ST(1). Результат z возвращается в ST(0), причем перед этим выполняется выталкивание значений x и y из стека сопроцессора. Команда FPATAN широко применяется для вычисления значений обратных тригонометрических функций, таких как arcsin, arccos, arcctg, arccosec, arcsec. Например, для вычисления функции arcsin используется следующая формула:

$$\arcsin(a) = \mathrm{arctg}\left(\frac{a}{\sqrt{1-a^2}}\right).$$

Для вычисления этой формулы необходимо выполнить следующую последовательность шагов.

1. Если a является мерой угла в градусах, то выполнить ее преобразование в радианную меру по правилу, приведенному ранее.
2. Поместить в стек a в радианной мере.
3. Вычислить значение выражения $\sqrt{1 - a^2}$ и поместить его в стек.
4. Выполнить команду FPATAN с аргументами в регистрах ST(0) и ST(1). В ST(0) должно быть значение выражения $\sqrt{1 - a^2}$, в st(l) — a.

В результате этих действий в регистре ST(0) будет сформировано значение выражения $\arcsin(a)$.

Аналогично вычисляются значения других обратных тригонометрических функций.

Для вычисления функции $\arccos(a)$ используется формула

$$\arccos(a) = 2\ \mathrm{arctg}\left(\frac{\sqrt{1-a}}{\sqrt{1+a}}\right).$$

Для ее вычисления необходимо выполнить следующую последовательность шагов:

1. Если a является мерой угла в градусах, то выполнить ее преобразование в радианную меру по правилу, приведенному ранее
2. Вычислить значение выражения $\sqrt{1-a}$ и поместить его в стек.
3. Вычислить значение выражения $\sqrt{1+a}$ и поместить его в стек.
4. Выполнить команду FPATAN с аргументами в регистрах ST(0) и ST(1). В ST(0) должно быть значение выражения $\sqrt{1+a}$, в ST(1) — $\sqrt{1-a}$ •

В результате этих действий в регистре ST(0) сформируется значение выражения $\arccos(a)$.

Для вычисления $\text{arcctg}(a)$ используется формула

$$\text{arcctg}(a) = \text{arctg}(1/a).$$

Для ее вычисления необходимо выполнить следующую последовательность шагов:

1. Если a является мерой угла в градусах, то выполнить ее преобразование в радианную меру по правилу, приведенному ранее.
2. Командой FLD1 поместить в стек значение 1.
3. Поместить в стек значение a.
4. Выполнить команду FPATAN с аргументами в регистрах ST(0) и ST(1). В ST(0) должно быть значение a, в ST(1) — 1.

В результате этих действий в регистре ST(0) сформируется значение выражения $\text{arcctg}(a)$.

Если вам понадобится вычислить значения других тригонометрических функций, то ход ваших рассуждений должен быть аналогичен приведенному выше. Зная набор функций, поддерживаемых сопроцессором, вам следует искать формулы приведения (или выводить их самим), выражающие нереализованные в сопроцессоре функции посредством реализованных. Не забывайте контролировать диапазоны значений аргументов для тех команд, которые требуют этого.

Для демонстрации применения команд данной группы реализуем два алгоритма построения графических изображений на основе определенных математических зависимостей. Реализацией этих примеров мы завершим разработку Windows-приложения, о котором рассказывалось в главе 16. Если вы помните, это было приложение, в котором остались нереализованными два пункта меню: Графика ▸ Эффекты ▸ Павлин и Графика ▸ Эффекты ▸ Кружева. Рассмотрим вначале суть алгоритмов, в соответствии с которыми строятся эти графические изображения. Математические зависимости для построения этих фигур взяты из [17].

Фигура «Павлин» состоит из множества отрезков. Один конец каждого из этих отрезков располагается вдоль горизонтальной линии. Расположение второго конца рассчитывается по определенным формулам, содержащим тригонометрические функции sin и cos. Последовательность шагов построения фигуры «Павлин» выглядит следующим образом.

1. Организуется цикл по переменной xl (координата первого конца каждого отрезка), которая изменяется от 0 до значения icycl.
2. Для текущего значения xl вычисляются значения координат другого конца отрезка (x2, y2) по формулам

$$x2 = i120 + il00 \cdot \sin(xl/i30);$$
$$y2 = i90 + il00 \cdot \cos(xl/i30).$$

3. Строится отрезок, координаты концов которого располагаются в точках (xl, icenter) и (x2, y2).
4. Если значение xl не превысило значения icycl, то цикл повторяется для xl, увеличенного на 1.

В этом алгоритме значения icenter, icycl, i90, il00, i120, i30 представляют собой константы, изменяя которые/вы будете получать на экране другие варианты графического изображения «Павлин». На рис. 17.18 показан результат работы этого алгоритма.



Рис. 17.18. Результат работы приложения (пункт меню Графика ▸ Эффекты ▸ Павлин)

Несущественные детали алгоритма вы легко восстановите по фрагменту программы, приведенному в листинге 17.9. Полный текст программы приведен в каталоге данной главы среди файлов, прилагаемых к книге[1].

Листинг 17.9. Фрагмент программы для воспроизведения фигуры «Павлин»

```
.data
...
;определение констант для фигуры "Павлин"
xl  dd  1
x2  dd  0
y2  dd  0
```

[1] Все прилагаемые к книге файлы можно найти по адресу http://www.piter.com/download. — *Примеч. ред.*

```
i30     dw  30
i90     dw  90
1100    dw  100
i120    dw  120
icenter dd  100
icycl   dd  319
...
.code
...
;-------------------MenuProc---------------------------------------------
;обработка сообщений от меню
MenuProc    proc
...
@@idmpeacock:   ;"Павлин"
;очистим окно
;выполним первичное заполнение растра серым цветом
;получим дескриптор серой кисти hbrush=GetStockObject(GRAY_BRUSH)
    push    GRAY_BRUSH
    call    GetStockObject
    mov     @@hbrush, eax
;выбираем кисть в контекст памяти SelectObject(memdc, @@hbrush)
    push    @@hbrush
    push    memdc
    call     SelectObject
;заполняем выбранной кистью виртуальное окно
;BOOL PatBlt(HDC hdc, int nXLeft, int nYLeft, int nWidth, int nHeight, DWORD
dwRop)
    push    PATCOPY
    push    raaxY
    push    maxX
    push    NULL
    push    NULL
    push    memdc
    call    PatBlt
    mov     ecx, icycl
    push    ecx
@@m1:
    finit
;вычислим x2=120+100*sin(x1/30)
    pop     ecx
    mov     x1, ecx
    cmp     ecx,0
    je      @@m2
    dec     ecx
    push    ecx
    fild    x1
    fidiv   i30
    fsin
    fimul   i100
    fiadd   i120
    fistp   x2
;вычислим y2=120+100*cos(x1/30)
    fild    x1
    fidiv   i30
    fcos
    fimul   i100
    fiadd   i90
    fistp   y2
;рисуем отрезок:
    push    NULL
    push    icenter
    push    x1
    push    memdc
    call    MoveToEx
```

продолжение ➭

Листинг 17.9 *(продолжение)*

```
    push    y2
    push    x2
    push    memdc
    call    LineTo
;генерация сообщения WM_PAINT для вывода строки на экран
    push    0
    push    NULL
    push    @@hwnd
    call    InvalidateRect
    jmp     @@m1
@@m2:
    jmp     @@exit
```

Фигура «Кружева» формируется следующим образом. В окне приложения строятся вершины правильного многоугольника, количество которых задается константой N. Каждая из этих N вершин соединяется отрезками с другими вершинами. Координаты вершин многоугольника задаются формулами

$$xi = xc + r \cdot \cos(2\pi/N) \cdot icenter;$$
$$yi = yc + r \cdot \sin(2\pi/N).$$

Здесь i — номер вершины; r — радиус описанной около многоугольника окружности; (xc, yc) — координаты центра многоугольника.

В этом алгоритме значения N, r, xc, yc представляют собой константы, изменяя которые, вы будете получать на экране другие варианты графического изображения «Кружева». На рис. 17.19 показан результат работы программы.

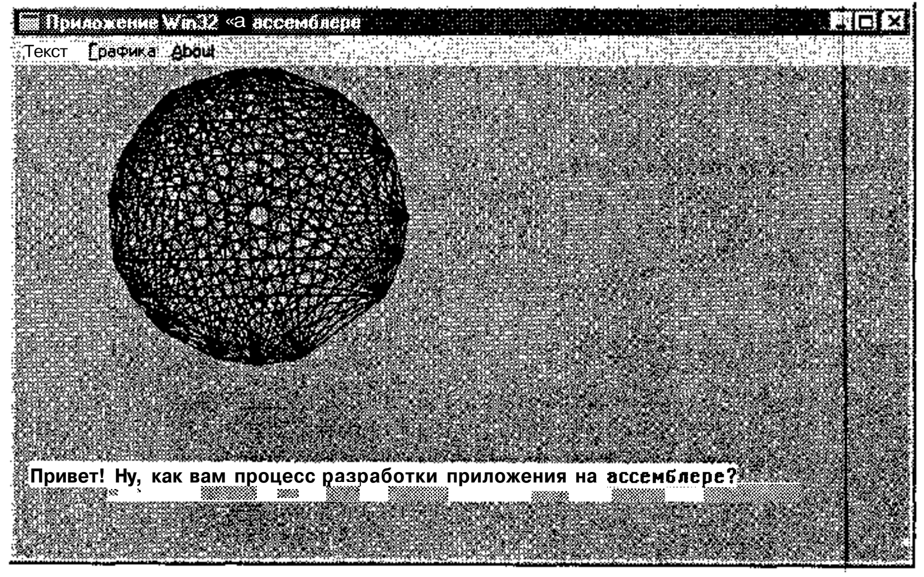


Рис. 17.19. Результат работы приложения (пункт меню Графика ▸ Эффекты ▸ Кружева)

Несущественные детали алгоритма вы легко восстановите по фрагменту программы, приведенному в листинге 17.10. Полный текст программы можно найти среди файлов, прилагаемых к книге, в каталоге данной главы.

Листинг 17.10. Фрагмент программы для изображения фигуры «Кружева»

```
.data
...
;определение констант для фигуры "Кружева"
;N - число вершин правильного многоугольника
;его можно менять - попробуйте!!!
N   equ 18
Xc  equ 160
Yc  equ 100
masX    dd  N dup (0)
masY    dd  N dup (0)
iN  dw  N
R   dw  99
DTT dd  0
t   dd  0
i   dd  0
j   dd  0
i2  dw  2
...
.code
...
;--------------------MenuProc----------------------------------------
;обработка сообщений от меню
MenuProc    proc
...
@@idmlaces: ;"Кружева"
;очистим окно
;выполним первичное заполнение растра серым цветом
;получим дескриптор серой кисти hbrush=GetStockObject(GRAY_BRUSH)
    push    GRAY_BRUSH
    call    GetStockObject
    mov     @@hbrush, eax
;выбираем кисть в контекст памяти SelectObject(memdc, @@hbrush)
    push    @@hbrush
    push    memdc
    call    SelectObject
;заполняем выбранной кистью виртуальное окно
;BOOL PatBlt(HDC hdc, int nXLeft, int nYLeft, int nWidth,
;   int nHeight, DWORD dwRop)
    push    PATCOPY
    push    maxY
    push    maxX
    push    NULL
    push    NULL
    push    memdc
    call    PatBlt
;вычислим DTT=2*pi/N
    finit
    fld     Pi
    fidiv   iN
    fimul   i2
    fistp   DTT
    mov     t, 0
    mov     i, 0
;заполняем массивы masX и masY координатами вершин прямоугольника
@@m3:
    mov     eax, i
    add     eax, DTT
    mov     t, eax
    fild    t
    fcos
    fimul   R
    mov     esi , i
```

продолжение

Листинг 17.10 *(продолжение)*

```
    fistp   masX[esi*4]
    add     masX[esi*4], Xc
    fild    t
    fsin
    fimul   R
    fistp   masY[esi*4]
    mov     eax, Yc
    sub     eax, masY[esi *4]
    mov     masY[esi*4], eax
    inc     i
    cmp     i, N
    jl      @@m3
;соединим вершины многоугольника:
    mov     i, 0
@@m5:
    mov     eax, i
    mov     j, eax
@@m4:  inc j
;рисуем отрезок:
    push    NULL
    mov     esi, i
    push    masY[esi*4]
    push    masX[esi*4]
    push    memdc
    call    MoveToEx
    mov     edi , j
    push    masY[edi*4]
    push    masX[edi*4]
    push    memdc
    call    LineTo
    cmp     j, N
    jl      @@m4
    inc     i
    cmp     i, N
    jl      @@m5
;генерация сообщения WM_PAINT для вывода строки на экран
    push    0
    push    NULL
    push    @@hwnd
    call    InvalidateRect
    jmp     @@exit
```

Отметим еще две команды сопроцессора, FPREM и FPREM1.

Команда FPREM — команда получения *частичного* остатка от деления. Исходные значения делимого и делителя размещаются в стеке — делимое в ST(0), делитель в ST(1). Делитель рассматривается как некоторый *модуль.* Поэтому в результате работы команды получается остаток от деления по модулю. Но произойти это может не сразу, так как этот результат в общем случае достигается за несколько последовательных обращений к команде FPREM, если значения операндов сильно различаются. Физически работа команды заключается в реализации хорошо известного всем действия: деления в столбик. При этом каждое промежуточное деление осуществляется отдельной командой FPREM. Цикл, центральное место в котором занимает команда FPREM, завершается, когда очередная полученная разность в ST(0) становится меньше значения модуля в ST(1). Судить об этом можно по состоянию флага c2 в регистре состояния SWR:

* если C2 = 0, то работа команды FPREM полностью завершена, так как разность в ST(0) меньше значения модуля в ST(1);

- если C2 = 1, то необходимо продолжить выполнение команды FPREM, так как разность в ST(0) больше значения модуля в ST(1).

Таким образом, необходимо анализировать флаг C2 в теле цикла. Для этого C2 записывается в регистр флагов основного процессора с последующим анализом его командами условного перехода. Другой способ заключается в сравнении ST(0) и ST(1).

Необходимость в таком частичном исполнении команды FPREM возникает из-за того, что если делимое слишком велико, а делитель мал, то время получения конечного частичного остатка, удовлетворяющего условию ST(0) < ST(1), может быть достаточно большим. При этом становится невозможной своевременная реакция на поступающие запросы прерываний, возможно, достаточно важные для того, чтобы быть задержанными в обработке.

В контексте рассмотрения нами трансцендендентных команд пример использования команды FPREM является наиболее показательным. При рассмотрении команды FPTAN мы упомянули о том, что аргумент z функции tg должен находиться в диапазоне $0 < z < \pi/4$. Исходя из того, что данная тригонометрическая функция является периодической, возникает необходимость понижения размерности аргумента z, находящегося за рамками указанного диапазона допустимых значений Для команды FPTAN, что и делается с помощью команды FPREM.

Важно отметить, что команда FPREM не соответствует последнему стандарту IEEE-754 на вычисления с плавающей точкой. По этой причине в систему команд сопроцессора i387 была введена команда FPREM1, которая отличается от команды FPREM тем, что накладывается дополнительное требование на значение остатка в ST(0). Это значение не должно превышать половины модуля в ST(1). В остальном работа команды FPREM1 аналогична работе FPREM.

Команды FPREM и FPREM1 имеют еще одну особенность, представляющую интерес для команд, вычисляющих значения периодических тригонометрических функций. После полного завершения работы команды FPREM/FPREM1 (когда C2 = 0) биты C0, C3, C1 содержат значения трех младших битов частного, которые логически представляют собой численное значение номера одного из восьми октантов единичного круга. Это, несомненно, важная информация при работе с тригонометрическими функциями.

Далее рассмотрим еще три трансцендентные команды.

Команда F2XM1 — команда вычисления значения функции $y = 2^x - 1$. Исходное значение x размещается в вершине стека сопроцессора в регистре ST(0) и должно лежать в диапазоне $-1 < x < 1$. Результат y замещает x в регистре ST(0). Эта команда может быть использована для вычисления различных показательных функций (например, далее показано, как она используется для возведения числа в степень).

Единица вычитается для того, чтобы получить точный результат, когда значение x близко к нулю. Вспомните, что нормализованное число всегда содержит в качестве первой значащей цифры единицу. Поэтому, если в результате вычисления функции T получается число 1,000000000456..., то команда F2XM1, вычитая 1 из этого числа и затем нормализуя результат, формирует больше значащих цифр, то есть делает его более точным. Неявное вычитание единицы командой F2XM1 компенсируется командой FADD с единичным операндом.

Команда FYL2X — команда вычисления значения функции $2 = y \log_2(x)$. Исходное значение x размещается в вершине стека сопроцессора, а исходное значение y — в регистре ST(1). Значение x должно лежать в диапазоне $0 < x < +\infty$, а значение y — в диапазоне $-\infty < y < +\infty$. Перед тем как осуществить запись результата z в вершину стека, команда FYL2X выталкивает значения x и y из стека и только после этого производит запись z в регистр ST(0).

Команды сопроцессора F2XM1 и FYL2X очень полезны, в частности, для реализации операции возведения в степень любого числа по любому основанию. Специальной команды в сопроцессоре для возведения в степень нет. Поэтому программисту нужно искать подходящую формулу приведения, которая позволит реализовать отсутствующую операцию имеющимися программно-аппаратными средствами сопроцессора. Возведение в произвольную степень числа с любым основанием производится по формуле:

$$x^y = 2y \cdot \log_2(x).$$

Вычисление значения выражения $y \cdot \log_2(x)$ для любого y и $x > 0$ производится специальной командой сопроцессора FYL2X, рассмотренной ранее. Вычисление I^x производится командой F2XM1 (лишнее вычитание единицы пока не замечаем, так как его всегда можно компенсировать сложением с единицей). Но в последнем действии есть тонкий момент, который связан с тем, что величина аргумента x должна лежать в диапазоне: $-1 \le x \le 1$. А как быть в случае, если x превышает это значение (например, 16^3). При вычислении выражения $3 \cdot \log_2(16)$ командой FYL2X вы получите в стеке значение 12. Попытка вычислить значение 2^{12} командой F2XM1 ни к чему не приведет — результат окажется неопределенным (по моим наблюдениям, значение ST(0) попросту не изменяется). В этой ситуации логично вспомнить о другой команде сопроцессора, FSCALE, которая также вычисляет значение выражения T, но для целых значений x со знаком. Применив формулу $2^{a+b} = 2^a \cdot 2^b$, получаем решение проблемы. Разделяем дробный показатель степени, больший $|1|$, на две части — целую и дробную. После этого вычисляем отдельно командами FSCALE и F2XM1 степени двойки и перемножаем результаты (не забываем компенсировать вычитание единицы в команде F2XM1 последующим сложением результата с единицей).

Далее представлен обобщенный алгоритм вычисления степени произвольного числа x по произвольному показателю y (он не претендует на оптимальность решения; его основное назначение — раскрытие принципов реализации этой операции).

1. Загрузить в стек основание степени x.
2. Загрузить в стек показатель степени y и проверить его знак. Если $y < 0$, то запомнить этот факт (для шага 8) и заменить в стеке значение y его модулем (это следует из формулы $x^{-y} = 1/x^y$).
3. Командой FYL2X вычислить значение выражения $z = y \lg_2(x)$.
4. Проверить z:
   - если значение $|z| < 1$, то обозначить z как z2;
   - если значение $|z| > 1$, то представить z в виде двух слагаемых $z1 + z2$, где z1 — целое значение, а z2 — значение меньше единицы (например, число 16,84

следует представить в виде суммы 16 + 0,84); в программе это можно выполнить путем циклического вычитания единицы из начального значения г до тех пор, пока оно не станет меньше единицы (количество вычитаний нужно запомнить, обозначив через п).

5. Выполнить команду F2XM1 с аргументом z2.
6. Выполнить команды FLD1 и FADD, компенсирующие единицу, которая была вычтена из результата функции 2^x на шаге 5 командой F2XM1.
7. Выполнить команду FSCALE с аргументом zl. Переменную z1 нужно инициализировать нулем, что позволит получить корректный результат, даже если исходное значение *z* было меньше единицы.
8. Перемножить результаты, полученные на шагах 6 и 7, — в вершине стека окажется результат возведения в степень. Если показатель степени, проверяемый на шаге 2, был отрицательным, необходимо единицу разделить на этот результат. Для этого в стек загружаем единицу командой FLD1 и применяем команду FDIV.

Данный алгоритм реализует программа листинга 17.11.

Листинг 17.11. Возведение числа в произвольную степень

```
.586p
masm
model   use16 small
.stack  100h
.data              ;сегмент данных
flag    db  0
p1  dd  0
y   dt  2.0        ;основание степени
x   dt  -2.0       ;показатель степени
.code
main    proc       ;начало процедуры main
    mov ax, @data
    mov ds, ax
    finit
    fld y
    fld x
    ftst
    fstsw   ax
    sahf
    jnc ml         ;переход, если x >=0
    inc flag       ;взведем flag, если x<0
    fabs           ;|x|
ml: fxch
    fyl2x
    fst st(1)
    fabs           ;|z|
;сравним |z| с единицей:
    fld1
    fcom
    fstsw   ax
    sahf
    jp  exit       ;операнды не сравнимы
    jnc m2         ;если |z|<1, ТО переход на m2
    jz  m3         ;если |z|=1, то переход на m3
                   ;если |z|>1, то приводим к формуле z=zl+z2, где zl - целое,
                   ;z2 f дробное и z2<1:
    xor ecx, ecx;счетчик вычитаний
```

продолжение

Листинг 17.11 *(продолжение)*

```
m12:    inc cx
    fsub    st(1),  st(0)
    fcom
    fstsw   ax
    sahf
    jp  exit;операнды не сравнимы
    jz  m12
    jnc    ra2    ;если |z|<1, то переход на m2
    jmp m12      ;если |z|>1, то переход на m12
m3: mov p1, 1
    jmp $+7
m2: mov p1, ecx
    fxch
    f2xm1
    fadd         ;компенсируем 1
    fild p1      ;показатель степени для fscale
    fld1
    fscale
    fxch
    fincstp
    fmul
;проверка на отрицательную степень
    cmp flag, 1
    jnz exit
    fld1
    fxch
    fdiv
exit:
    mov ax, 4c00h
    int 21h
main    endp
end main
```

Используя тождество $\log_n(x) - \log_n(2) \cdot \log_2(x)$, можно вычислить логарифм любого числа по любому основанию. Значения $\log_n(2)$, $\log_2(10)$ и некоторых других логарифмических функций вычисляются соответствующими командами сопроцессора: FLDL2T, FLDL2E, FLDLG2, FLDLN2.

Еще одна команда FYL2XP1 — команда вычисления значения функции $z = y \log_2(x + 1)$. Исходное значение x размещается в вершине стека сопроцессора — регистре ST(0), а исходное значение y — в регистре ST(1). Значение x должно лежать в диапазоне $0 < |x| < 1 - 1/\sqrt{2}$, а значение y — в диапазоне $-\infty < y < +\infty$. Перед тем как записать результат z в вершину стека — регистр ST(0), команда FYL2XP1 выталкивает из стека значения x и y.

Очень интересную программу можно найти среди файлов, прилагаемых к книге, в подкаталоге ..\OutFPU каталога данной главы. Программа преобразует содержимое вершины стека в символьный вид и выводит его на экран. В этом примере иллюстрируется работа многих уже рассмотренных команд.

Команды управления сопроцессором

Последняя группа команд предназначена для общего управления работой сопроцессора. Команды этой группы имеют особенность — перед началом своего выполнения они не проверяют наличие незамаскированных исключений. Однако такая проверка может понадобиться, в частности, для того, чтобы при параллельной работе основного процессора и сопроцессора предотвратить разрушение информа-

ции, необходимой для корректной обработки исключений, возникающих в сопроцессоре. Поэтому некоторые команды управления имеют аналоги, выполняющие те же действия плюс одну дополнительную функцию — проверку наличия исключения в сопроцессоре. Эти команды имеют одинаковые мнемокоды (и машинные коды тоже), различающиеся только вторым символом — символом n:

- мнемокод, не содержащий второго символа n, обозначает команду, которая перед началом своего выполнения проверяет наличие незамаскированных исключений;
- мнемокод, содержащий второй символ n, обозначает команду, которая перед началом своего выполнения не проверяет наличия незамаскированных исключений, то есть выполняется немедленно, что позволяет сэкономить несколько машинных тактов.

Как уже упоминалось, эти команды имеют одинаковый машинный код. Отличие лишь в том, что перед командами, не содержащими символа n, транслятор ассемблера вставляет команду WAIT. Команда WAIT является полноценной командой основного процессора, и ее при необходимости можно указывать явно. Команда WAIT имеет аналог среди команд сопроцессора — FWAIT. Обеим этим командам соответствует код операции 9bh.

Команда WAIT/FWAIT — это команда ожидания. Она предназначена для синхронизации работы процессора и сопроцессора. Так как основной процессор и сопроцессор работают параллельно, то может создаться ситуация, когда за командой сопроцессора, изменяющей данные в памяти, следует команда основного процессора, которой эти данные требуются. Чтобы синхронизировать работу этих команд, необходимо включить между ними команду WAIT/FWAIT. Встретив данную команду в потоке команд, основной процессор приостановит свою работу до тех пор, пока не поступит аппаратный сигнал о завершении очередной команды в сопроцессоре. Здесь есть еще один эффект, отмеченный ранее. Он касается корректной обработки исключений и связанной с ними информации.

Далее с необходимой степенью подробности будут рассмотрены команды, входящие в группу команд управления. Некоторые из этих команд мы уже применяли в программах этой главы.

Из всех команд управления первой логично рассмотреть команду, приводящую сопроцессор в некоторое начальное состояние, — это команда инициализации сопроцессора FINIT/FNINIT. Она инициализирует управляющие регистры сопроцессора определенными значениями.

- Регистр управления CWR инициализируется числом 037h, что означает установку следующих режимов:
  - поле округления RC = 00 — округление к ближайшему целому;
  - биты 0...5 установлены в единицу, что означает маскирование всех исключений;
  - поле управления точностью PC = 11 — максимальная точность (64 бита).
- Регистр состояния SWR инициализируется нулевым значением, что означает отсутствие исключений и указание на то, что физический регистр стека сопроцессора R0 является вершиной стека и соответствует логическому регистру ST(0).

- Регистр тегов TWR инициализируется единичным значением — это означает, что все регистры стека сопроцессора пусты.
- Регистры указателей данных DPR и команд IPR инициализируются нулевыми значениями.

Данную команду используют перед первой командой сопроцессора в программе и в других случаях, когда необходимо привести сопроцессор в начальное состояние.

Следующие две команды работают с регистром состояния SWR.

- FSTSW/FNSTSW ax — команда сохранения содержимого регистра состояния SWR в регистре AX. Эту команду целесообразно использовать для подготовки к условным переходам по описанной при рассмотрении команд сравнения схеме.
- FSTSW/FNSTSW приемник — команда сохранения содержимого регистра состояния SWR в ячейке памяти. От рассмотренной ранее команда отличается типом операнда — теперь это ячейка памяти размером два байта (в соответствии с размерностью регистра SWR).

Обратите внимание на то, что команды, работающие с регистром SWR, выполняют только считывание содержимого этого регистра. Запись не предусмотрена, так как она попросту не требуется за исключением, возможно, поля TOP, которое является указателем текущей вершины стека. Для изменения этого поля предусмотрены отдельные команды FINCSTP, FDECSTP, которые будут рассмотрены далее. Следующие две команды, работающие с информацией в регистре управления CWR, напротив, поддерживают действие записи и чтения содержимого этого регистра.

- FSTCW/FNSTCW приемник — команда сохранения содержимого регистра управления CWR в ячейке памяти размером два байта. Эту команду целесообразно использовать для анализа полей маскирования исключений, управления точностью и округления. Следует заметить, что операндом не является регистр AX, в отличие от команды FSTSW/FNSTSW.
- FLDCW источник — команда загрузки значения ячейки памяти размером 16 битов в регистр управления CWR. Эта команда выполняет действие, противоположное действию команды FSTCW/FNSTCW. Команду FLDCW целесообразно использовать для задания или изменения режима работы сопроцессора. Следует отметить, что если в регистре состояния SWR установлен любой бит исключения, то попытка загрузки нового содержимого в регистр управления CWR приведет к возбуждению исключения. По этой причине необходимо перед загрузкой регистра CWR сбросить все флаги исключений в регистре SWR.

Следующая команда — команда без операндов FCLEX/FNCLEX — позволяет сбросить флаги исключений в регистре состояния SWR сопроцессора, которые, в частности, необходимы для корректного выполнения команды FLDCW. Ее также применяют в случаях, когда необходимо сбрасывать флаги исключений в регистре SWR, например, в конце подпрограмм обработки исключений. Если этого не делать, то при исполнении первой же команды сопроцессора прерванной программы (кроме тех команд, которые имеют в названии своего мнемокода второй символ л) опять возбуждается исключение.

Как мы отметили ранее, сопроцессор имеет две команды, которые работают с указателем стека в регистре SWR.

- FINCSTP — команда увеличения указателя стека на единицу (поле TOP) в регистре SWR. Команда не имеет операндов. Действие команды FINCSTP подобно действию команды FST, но она извлекает значение операнда из стека «в никуда». Таким образом, эту команду можно использовать для выталкивания ставшего ненужным операнда из вершины стека. Команда работает только с полем TOP и не изменяет соответствующее данному регистру поле в регистре тегов TWR, то есть регистр остается занятым и его содержимое из стека не извлекается.
- FDECSTP — команда уменьшения указателя стека (поле TOP) в регистре SWR. Команда не имеет операндов. Действие команды FDECSTP подобно действию команды FLD, но она не помещает значения операнда в стек. Таким образом, эту команду можно использовать для проталкивания внутрь стека операндов, ранее включенных в него. Команда работает только с полем TOP и не изменяет соответствующее данному регистру поле в регистре тегов TWR, то есть регистр остается пустым.

Следующая команда, FFREE st(i), помечает любой регистр стека сопроцессора как пустой. Это команда освобождения регистра стека ST(i). Команда записывает в поле регистра тегов, соответствующего регистру ST(i), значение 11b, что соответствует пустому регистру. При этом указатель стека (поле TOP) в регистре SWR и содержимое самого регистра не изменяются. Применяя эту команду, помните, что ST(i) — это относительный, а не физический номер регистра стека сопроцессора. Необходимость в этой команде может возникнуть при попытке записи в регистр ST(i), который помечен как непустой. В этом случае будет возбуждено исключение. Для предотвращения этого применяется команда FFREE.

В сопроцессоре есть также команда FNOP, не производящая никаких действий и влияющая только на регистр указателя команды IPR.

В группе команд управления можно выделить подгруппу команд, работающих с так называемой средой сопроцессора. *Среда сопроцессора* — это совокупность регистров сопроцессора и их значений. Среда сопроцессора может быть *частичной* или *полной*. О какой именно среде идет речь, определяется одной из рассмотренных далее команд:

- FSAVE/FNSAVE приемник — команда сохранения полного состояния среды сопроцессора в память, адрес которой указан операндом приемник. Размер области памяти зависит от размера операнда сегмента кода (use16 или use32):
  - use16 — область памяти должна составлять 94 байта (рис. 17.20, *а, б*): 80 байтов для восьми регистров из стека сопроцессора и 14 байтов для остальных регистров сопроцессора с дополнительной информацией;
  - use32 — область памяти должна составлять 108 байтов (рис. 17.20, *в, г*): 80 байтов для восьми регистров из стека сопроцессора и 28 байтов для остальных регистров сопроцессора с дополнительной информацией;
- FRSTOR источник — команда восстановления полного состояния среды сопроцессора из области памяти, адрес которой указан операндом источник. Сопроцессор будет работать в новой среде сразу после окончания работы команды FRSTOR.

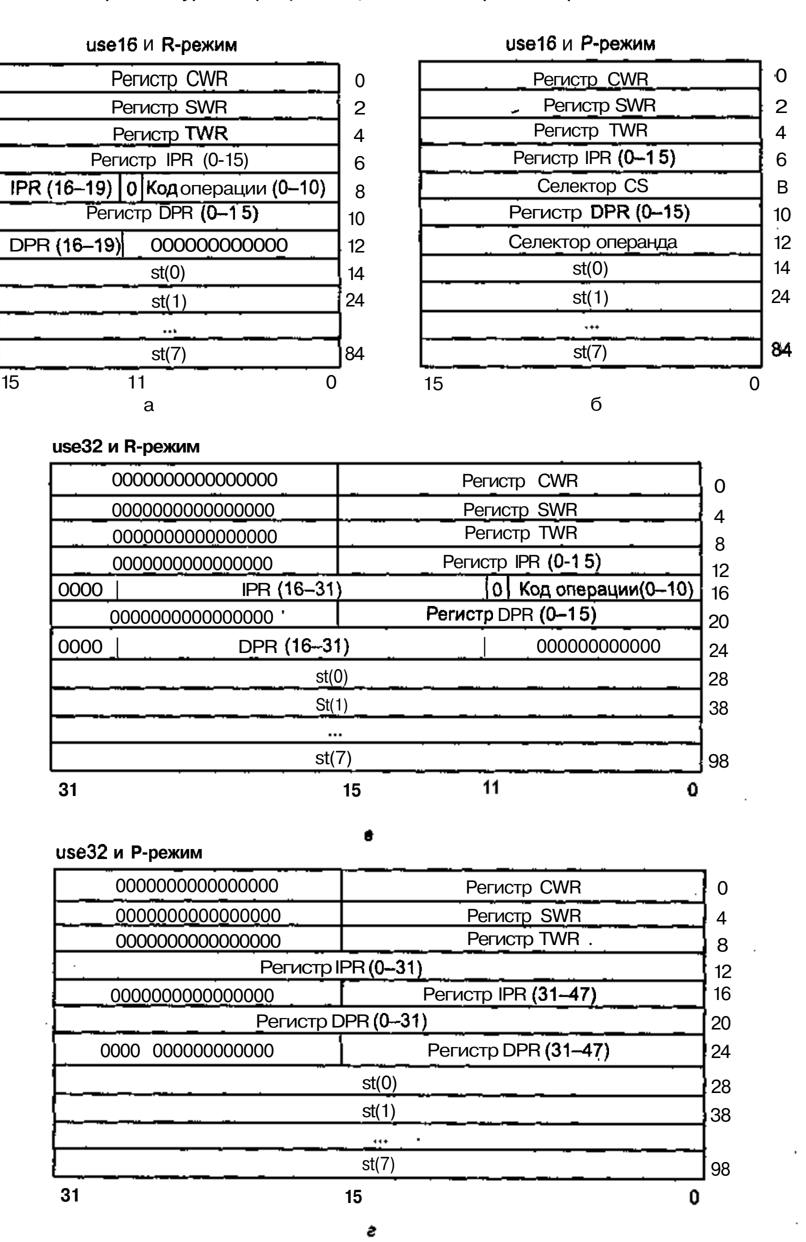


Рис. 17.20. Структура области памяти с полной средой сопроцессора в различных режимах

Следующие две команды сохраняют частичную среду сопроцессора, в состав которой входят регистры SWR, CWR, TWR, DPR и IPR.

- FSTENV/FNSTENV приемник — команда сохранения частичного состояния среды сопроцессора в область памяти, адрес которой указан операндом приемник. Размер области памяти зависит от размера операнда сегмента кода (use16 или use32).

Формат области частичной среды сопроцессора совпадает с форматом области полной среды (рис. 17.21), за исключением содержимого стека сопроцессора (80 байт).

- FLDENV источник — команда восстановления частичного состояния среды сопроцессора содержимым из области памяти, адрес которой указан операндом источник. Информация в данной области памяти была ранее сохранена командой FSTENV/FNSTENV.

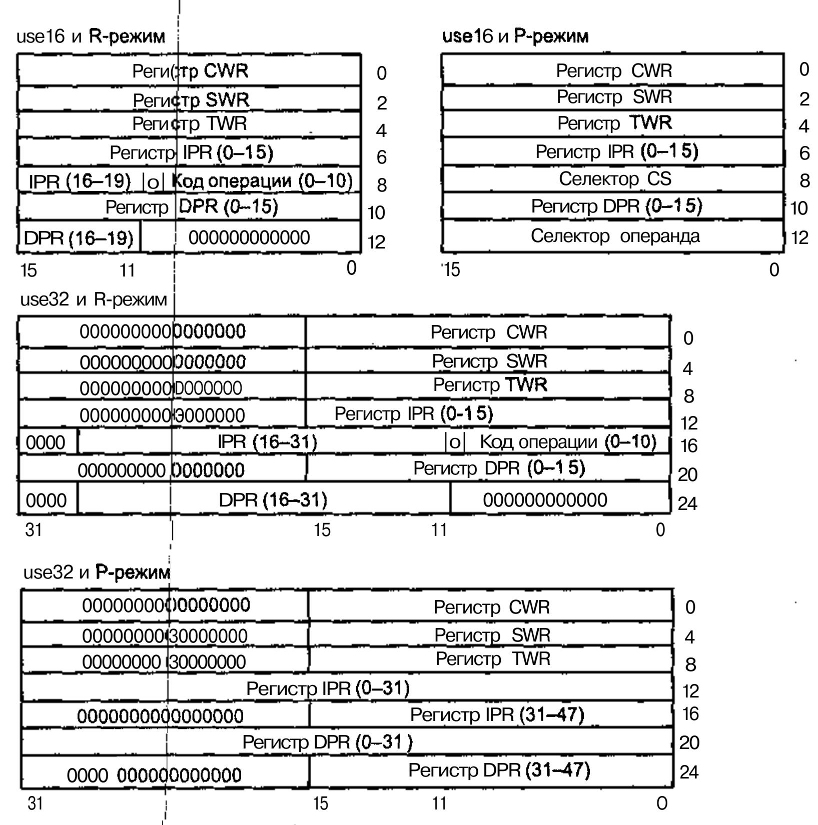


Рис. 17.21. Структура области памяти с частичной средой сопроцессора в различных режимах

Команды сохранения среды целесообразно применять в обработчиках исключений, так как только с помощью данных команд можно получить доступ, например, к регистрам DPR и IPR. Не исключено использование этих команд в подпрограммах или в многозадачной среде для переключения контекстов программ. Области памяти, содержащие сохраненные состояния среды сопроцессора, есть смысл располагать в стеке основной программы. Тогда, в частности, появляется возможность передать информацию коду, находящемуся в другом кольце защиты основного процессора.

Судя по последним командам, сопроцессор может работать не только в реальном, но и в защищенном режиме. Для этого необходимо выполнять переключение

сопроцессора между этими режимами. Операция переключения реализуется специальными командами.

- FSETPM — команда переключения сопроцессора из реального в защищенный режим. Команда не имеет операндов. Действие команды влияет только на выполнение команд сохранения и восстановления среды. Для реального и защищенного режимов состав и формат информации среды сопроцессора несколько различается.
- FRSTPM — команда переключения сопроцессора из защищенного в реальный режим. Команда не имеет операндов. Действие команды влияет только На выполнение команд сохранения и восстановления среды.

Исключения сопроцессора и их обработка

Важной особенностью сопроцессора является возможность распознавания особых ситуаций, которые могут возникать в процессе вычислений. При этом сопроцессор формирует определенные управляющие сигналы и устанавливает бит в своем регистре состояния SWR. Согласно идеологии IA-32, прерывания классифицируются по источникам возникновения. Один из возможных типов прерываний — *исключения,* которые являются внутренними прерываниями процессора, возникающими в ходе вычислительного процесса. Особые ситуации, возникающие в сопроцессоре, полностью соответствуют определению понятия «исключение».

В сопроцессоре могут возникать шесть типов исключений. Все они были перечислены при обсуждении форматов регистра состояния SWR и регистра управления CWR. Вспомним основные моменты. В регистре состояния SWR 6 битов, каждый из которых играет роль флага для определенного типа исключения. При возникновении исключения устанавливается соответствующий флаг в этом регистре (см. рис. 17.2). Сопроцессор аппаратно позволяет запретить явную обработку любого из этих типов исключений. Для этого в регистре управления CWR есть *6* битов, играющих роль масок, установка которых и позволяет запретить возникновение соответствующих исключений. Важно отметить, что запрещение обработки исключения вовсе не означает того, что сопроцессор оставляет вычислительную ситуацию неизменной. При возникновении исключения, имеющего единичное состояние соответствующего бита в регистре CWR, сопроцессор выполняет так называемую *маскированную реакцию,* которая включает в себя некоторую последовательность предопределенных разработчиками процессора действий. Естественно, что в них невозможно было предусмотреть все потребности программистов, которым, помимо всего прочего, могут понадобиться и нестандартные варианты реакции на исключения. В этом случае программисту необходимо установить соответствующий бит в регистре CWR. О том, как нужно реагировать на возникновение незамаскированного исключения, мы поговорим чуть позже, а пока же разберемся с причинами возникновения исключений и маскированными реакциями на них со стороны сопроцессора. Эта информация может быть полезна для того, чтобы разобраться с логическими ошибками программы и правильно запрограммировать реакцию на их возникновение.

Недействительная операция

Причиной исключения недействительной операции являются ошибки в логике программы, наиболее типичные из которых следующие:

- загрузка операнда в непустой регистр стека сопроцессора;
- попытка извлечения операнда из пустого регистра стека сопроцессора;
- использование операнда с недопустимым для данной операции значением.

При возникновении этого исключения устанавливается флаг IE (Invalid Operation) в регистре SWR (см. рис. 17.2), маскируется оно битом IM регистра управления CWR (см. рис. 17.4). Исключение недействительной операции возникает при работе со стеком и при арифметических вычислениях. В каких именно операциях возникло это исключение, судят по содержимому бита SF (Stack Fault — ошибка стека) регистра состояния SWR. Если он установлен в единицу, это говорит о том, что исключение было вызвано ошибкой в работе стека сопроцессора; напротив, нулевое состояние бита SF говорит о том, что в команде встретился неверный операнд.

Маскированная реакция на исключение недействительной операции зависит от причины ошибки. Если она возникла в результате некорректной работы стека, то сопроцессор перезаписывает содержимое того регистра стека, обращение к которому вызвало исключение. При перезаписи в него заносится специальное численное значение — спокойное нечисло. Если ошибка возникла в результате недопустимого операнда, то в большинстве ситуаций в регистре стека сопроцессора возвращается также спокойное нечисло.

Кстати, исключение недействительной операции — это единственное исключение, которое не нужно маскировать. Программисту следует самому вмешиваться в обработку ошибочных ситуаций путем вызова соответствующего обработчика.

Деление на ноль

Исключение деления на ноль возникает в командах, которые явно или неявно выполняют деление. Это команда FDIV и ее варианты — FYL2X и FXTRACT. Факт возникновения этого исключения фиксируется флагом ZE (Zero Divide) регистра состояния сопроцессора SWR и при необходимости маскируется битом ZM регистра управления CWR. Маскированная реакция сопроцессора заключается в формировании результата в виде знаковой бесконечности.

Денормализация операнда

Исключение денормализованного операнда возникает, когда команда пытается выполнить операцию с денормализованным операндом. При этом устанавливается флаг DE (Denormalized Operand), который маскируется битом DM регистра управления CWR. Если это исключение замаскировано, то его возникновение приводит только к установке флага DE, после чего сопроцессор нормализует операнд и вычислительный процесс продолжается. Если исключение денормализованного операнда не замаскировано, то вызывается обработчик исключения, который позволяет произвести необходимую обработку ситуации.

Переполнение и антипереполнение

Ситуации переполнения и антипереполнения возникают в случаях, когда порядок результата слишком велик или слишком мал для формата приемника. При возникновении этих исключений в регистре SWR устанавливаются флаги OE (Overflow) и UE (Underflow). Эти исключения маскируются битами OM и UM регистра управления CWR. Исключения могут возникнуть при работе арифметических команд и команд, преобразующих формат операндов, таких как FST.

Маскированная реакция для ситуации переполнения состоит в формировании граничных (максимальных или минимальных) значений, представимых в сопроцессоре, или специального численного значения в виде знаковой бесконечности.

Немаскированная реакция на возникновение этих исключений зависит от того, где должен формироваться результат. Если приемник — память, то мантисса результата округляется, а порядок приводится к середине своего диапазона. Если приемник — ячейка памяти, то значение в ней не запоминается, при этом не изменяется и содержимое регистра стека.

Неточный результат

Исключение неточного результата возникает в случае, когда результат работы команды нельзя точно представить в формате приемника и его приходится округлять. Например, вычисление любой периодической дроби вроде 1/3 будет приводить к возникновению такого типа исключения. В какую сторону произошло округление, можно судить по значению бита C1:

- если C1 = 0, то результат был усечен;
- если C1 = 1, то результат был округлен в большую сторону.

При возникновении этих исключений в регистре SWR устанавливаются флаг PE (Precision). Исключение маскируется битом PM регистра управления CWR.

Немаскируемая обработка исключений

Как производить немаскируемую обработку исключений? Для этого необходимо установить в ноль те флаги в регистре CWR, которые соответствуют интересующим нас типам исключений. Далее нужно написать обработчик исключения, реализующий последовательность действий по корректировке ситуации, приведшей к исключению. Неясным остается вопрос о том, как передать управление обработчику исключения? Для ответа на него нужно разобраться с проблемой взаимодействия процессора и сопроцессора при возникновении исключения.

В главе 2 мы упоминали о существовании группы системных регистров процессора. В контексте нашего рассмотрения интерес представляет один из них — регистр CR0. Он имеет несколько битов, имеющих отношение к сопроцессору (табл. 17.3).

Таблица 17.3. Биты CR0, имеющие отношение к сопроцессору

Положение бита в CR0	Название бита	Назначение
1	MP (Math Present)	Сопроцессор присутствует. Бит должен быть установлен в 1

Положение бита в CR0	Название бита	Назначение
2	EM (Emulation Math)	Эмуляция сопроцессора. Когда EM = 1, выполнение любой команды сопроцессора вызывает исключение 7. В программах для микропроцессосоров i8086...i386 это позволяло иметь альтернативные фрагменты кода для выполнения одинаковых вычислений с использованием команд сопроцессора (EM = 0) и без них (EM = 1), то есть тогда, когда в конкретной конфигурации компьютера сопроцессор отсутствует
3	TS (Task Switched)	Задача переключена. Бит предназначен для согласования контекстов основного процессора и сопроцессора. Бит TS устанавливается в единицу при каждом переключении задачи. Состояние этого бита проверяется процессором, если очередной выбранной командой является команда сопроцессора. Если бит TS установлен в единицу, то процессор возбуждает исключение 7, обработчик которого выполняет необходимые действия, возможно, по сохранению или восстановлению контекста вычислений с плавающей точкой. С битом TS работает команда CLTS, которая устанавливает значение этого бита в 0
4	ET (Extension Type)	Тип расширения. Единичное значение этого бита означает поддержку инструкций сопроцессора
5	NE (Numeric Error)	Численная ошибка. Бит определяет способ обработки исключений сопроцессора: через сигнал внешнего прерывания или путем генерации исключения (см. далее)

Начиная с модели i486 процессор и сопроцессор размещаются в одном корпусе. Это упростило организацию взаимодействия между ними. Рассмотрим процессы, протекающие в компьютере при возникновении одного из шести перечисленных ранее исключений сопроцессора. При возникновении ситуации исключения сопроцессор устанавливает бит суммарной ошибки ES в регистре состояния SWR и формирует на одном из своих выходов сигнал ошибки. Этот сигнал ошибки одновременно воспринимается самим процессором, который генерирует исключение 10h, и в то же самое время, независимо от процессора, заводится на вход IRQ13 программируемого контроллера прерываний, обработчик которого вызывается через вектор прерывания 75h [8]. Таким образом, появление сигнала ошибки на выходе сопроцессора приводит к генерации в основном процессоре двух исключений с номерами 10h и 75h.

Исключение 10h является синхронным, так как вызов его обработчика санкционируется процессором при выполнении команд WAIT/FWAIT. Данные команды в процессорах i486 и Pentium встроены практически во все команды сопроцессора за исключением некоторых команд управления, поэтому работа любой команды сопроцессора начинается с выяснения того, было ли зафиксировано какое-нибудь

из незамаскированных исключений. Здесь важно то, что контекст вычислительной ситуации после выполнения команды, вызвавшей исключение, оказывается полностью сформированным (так как исключение синхронное, то есть ожидаемое) и с ним можно корректно работать.

Если сигнал поступает на вход программируемого контроллера прерываний IRQ13, то обработка исключения 75h может начаться в процессоре раньше, чем в сопроцессоре закончит выполняться команда, вызвавшая исключение, то есть в этом случае обработка прерывания является асинхронной к вычислительному процессу.

Необходимо отметить влияние бита NE на процессы, протекающие в компьютере при возникновении исключения. Его состояние определяет стиль обработки исключения процессором. Если бит NE = 1, то процессор возбуждает исключение 16 (обработка в стиле i286 и выше), если NE = 0, то при возникновении исключения процессор останавливается и ждет прерывания от программируемого контроллера прерываний (обработка в стиле i8086). По умолчанию бит NE устанавливается в 0.

Если посмотреть на распределение прерываний в реальном и защищенном режимах, то необходимо обратить внимание на номер 7 вектора прерываний — обращение к несуществующему сопроцессору. Прерывание появилось в процессоре i286, для которого сопроцессор не являлся обязательным устройством. Для того чтобы программа, выполняющая математические вычисления, была независимой от аппаратной конфигурации конкретного компьютера, писалось два варианта фрагментов кода, на которых эти вычисления выполнялись, — один с использованием команд сопроцессора и второй с использованием целочисленных команд. При распознавании в потоке команд инструкций сопроцессора процессоры i286 и i386 проверяли бит эмуляции сопроцессора EM (см. табл. 17.3). Если он был равен 1, то процессор возбуждал исключение 7. Это означало, что сопроцессора в конфигурации компьютера нет и его функции должны эмулироваться командами целочисленного устройства. Забота об установке бита EM ложилась на системное программное обеспечение.

Для процессоров i486 и Pentium состояние вычислительной среды определяется состоянием регистров после выполнения команды FINIT и содержимым регистра CR0 (биты MP и NE).

Что должен делать обработчик исключений сопроцессора? Его действия зависят от того, какое незамаскированное исключение им обрабатывается. Следует отметить основные действия по обработке любого исключения.

1. Сохранение среды сопроцессора командой FSTENV. Это необходимо для последующего выяснения причин исключения, а в среде сопроцессора как раз и зафиксировано состояние регистров управления сопроцессором при возникновении исключения.

2. Сброс установленных битов исключений в регистре SWR для предотвращения циклического возникновения исключений.

3. Выяснение типа исключения. Если незамаскированными являются исключения нескольких типов, то обработчик путем анализа соответствующих битов в регистре SWR должен определить их. Заметим, что содержимое SWR берется из сохраненной при входе в процедуру по команде FSTENV среды сопроцессора.

4. Выполнение действий по корректировке ситуации.
5. Возвращение в прерванную программу (командой IRET).

Однако выяснить причину исключения мало, да это и несложно. Важно, исправив ситуацию, вернуть управление прерванной программе. В защищенном режиме работы процессора прерывания (исключения) делятся на несколько групп: сбои, ловушки, исключения. Это деление осуществляется в зависимости от того, какая информация сохраняется о месте возникновения прерывания (исключения) и возможности возобновления прерванной программы. Для сопроцессора ситуация аналогична. Здесь информация о месте возникновения исключения зависит от типа исключения. Так, для исключений недействительной операции, деления на ноль и денормализованного операнда запоминается адрес команды, вызвавшей исключение. То есть ситуация для этих типов исключений распознается, и исключение возбуждается до исполнения команды сопроцессора (по классификации для защищенного режима — это сбой). При этом операнды в стеке и в памяти не модифицируются. Для остальных типов исключений ошибочная ситуация распознается после выполнения действий «виноватой» команды. Это означает, что в стеке в качестве адреса места возникновения исключения запоминается адрес следующей (после виновницы) команды программы. При этом операнды в памяти и в регистровом стеке, возможно, будут изменены. Ваши действия должны учитывать эти особенности возникновения различных типов исключений.

Использование отладчика

Отладчик Turbo Debuger предоставляет широкие возможности для отладки программ, использующих сопроцессор. Для наблюдения за состоянием регистров, составляющих программную модель сопроцессора в среде Turbo Debuger, необходимо открыть специальное окно Numeric processor. Для этого выберите пункт главного меню View ▸ Numeric processor или нажмите сочетание клавиш Alt+V и далее N. По умолчанию окно появится в компактном виде. Для того чтобы раскрыть его полностью, щелкните мышью на стрелке в правом верхнем углу окна.

В заголовке окна отображаются четыре сообщения.

- Модель сопроцессора (автоматически определяется отладчиком).
- «IPTR=...» — сообщение о текущем содержимом указателя команд. Этот указатель содержит физический (20-разрядный) адрес памяти, по которому расположена последняя выполнявшаяся инструкция сопроцессора.
- «OPTR=...» — сообщение об адресе памяти, к которому обращалась последняя команда сопроцессора (если она имела адресный операнд).
- «OPCODE=...» — сообщение о коде операции последней исполняемой команды сопроцессора. Интересно отметить то, как формирует отладчик код операции в этом поле. Мы отмечали, что машинный код операции всех команд сопроцессора начинается с одинаковой последовательности битов — 11011, поэтому в поле OPCODE эти биты отбрасываются. Например, код команды fld — 0d906h (в двоичном виде — 1101 1001 0000 ОНО). Убираем пять битов, одинаковых для кода операции каждой команды сопроцессора, и получаем то, что

видим в поле OPCODE заголовка окна Numeric processor, — 0106h (0000 0001 0000 0110).

В окне Numeric processor выделяются три области. Сразу заметим, что в отличие от областей окна CPU области окна Numeric processor нельзя раскрывать отдельно. Основную часть окна Numeric processor занимает область Registers, которая отражает состояние восьми регистров стека сопроцессора ST(0)...ST(7). Указываются только логические номера регистров. Наиболее полная информация о регистрах стека предоставляется, если окно Numeric processor развернуто. Рассмотрим поля Registers, описывающие состояние каждого из регистров стека сопроцессора. Первое поле показывает состояние регистра. Возможные значения в этом поле следующие:

- EMPTY – «пустой»;
- VALID — в регистре корректное вещественное число;
- ZERO — в поле нулевое значение;
- NaN — в регистре находится специальное численное значение — нечисло (Not a Number).

Второе поле показывает логический номер регистра стека. Третье поле содержит значения в регистре в виде 80-разрядного числа с плавающей точкой. Четвертое поле показывает содержимое регистра стека в шестнадцатеричном виде.

В ходе отладки вы можете влиять на содержимое регистров стека. Для этого в области Registers можно вызвать контекстное меню, активизируемое правой кнопкой мыши. В меню три команды:

- ZERO — обнуление содержимого регистра;
- EMPTY — освобождение регистра стека, при этом содержимое самого регистра стека не изменяется, а изменению подвергается только тег в регистре тегов, в который заносится значение 11b;
- CHANGE — запись в регистр стека некоторого значения, которое должно быть в допустимом формате в соответствии с синтаксисом ассемблера.

Следующую область окна Numeric processor условно можно назвать Control. Область Control содержит совокупность полей, названия которых совпадают с названиями битов или полей битов в регистре управления сопроцессором CWR. Перечислим эти поля:

- IM — маска недействительной операции;
- DM — маска денормализованного операнда;
- ZM — маска деления на нуль;
- OM — маска переполнения;
- UM — маска отрицательного переполнения;
- PM — маска точности;
- IEM — маска запроса на прерывание (для i8087);
- PC — поле управления точностью;
- RC — поле управления округлением;
- IC — поле управления значением бесконечности.

Контекстное меню области Control содержит всего одну команду — Toggle. Ее назначение — циклическое изменение содержимого активного (в котором находится курсор) поля.

Третья область окна Numeric processor — Status — содержит совокупность полей, названия которых совпадают с названиями битов или полей битов в регистре состояния сопроцессора SWR:

- IE — ошибка недействительной операции;
- DE — ошибка денормализованного операнда;
- ZE — ошибка деления на нуль;
- OE — ошибка переполнения;
- UE — ошибка отрицательного переполнения;
- PE — ошибка точности;
- IR — маска запроса на прерывание;
- CC — код условия (состояние битов C3, C2, C1, C0);
- ST — указатель вершины стека (поле TOP регистра SWR).

Контекстное меню области Status содержит всего одну команду — Toggle. Ее назначение — циклическое изменение содержимого активного поля.

Сам процесс отладки программы ничем не отличается от процесса отладки программы для основного процессора.

Общие рекомендации по программированию сопроцессора

В заключение главы сформулируем некоторые общие рекомендации по написанию программ для сопроцессора.

- Первый фрагмент программы с командами сопроцессора должен начинаться с команды FINIT. Если программа содержит несколько независимых друг от друга фрагментов с командами сопроцессора, каждый такой фрагмент должен начинаться с команды FINIT.
- При написании программы вы должны учитывать то, что процессор и сопроцессор работают параллельно. То есть вам необходимо особенно тщательно программировать участки, на которых планируется параллельное выполнение команд обоих процессоров. Особое внимание обращайте на синхронизацию общих операндов и обработку возможных исключительных ситуаций.
- Рекомендуется обработку исключений доверять самому сопроцессору, кроме исключения недействительной операции, что позволит своевременно обнаружить ошибки алгоритма.
- При написании программ следует установить такой режим округления, который позволит получить максимально точный результат.
- Для повышения производительности процессора при передаче данных необходимо использовать директиву EVEN. Ее действие заключается в том, что данные,

описываемые следующей за ней одной из директив резервирования и инициализации данных, размещаются по ближайшему адресу, значение которого кратно 2. Так как все типы данных сопроцессора имеют длину, кратную двум, то желательно все ячейки памяти, содержащие значения для обработки сопроцессором, размещать в сегменте данных одним блоком, предваряя их описание директивой EVEN, например:

```
...
.data
...
    even
ch_1    dd  35.78
ch_2    dt  0987687686
...
```

Итоги

- Математический сопроцессор значительно расширяет возможности компьютера по выполнению вычислений над числами из очень большого диапазона значений.
- Центром программной модели сопроцессора является регистровый стек, который является наиболее эффективной структурой программирования вычислительных алгоритмов. Использование стека предполагает, что программист предварительно преобразует исходное выражение в форму ПОЛИЗ. Форма ПОЛИЗ, в частности, используется в трансляторах при разборе и генерации кода различных синтаксических конструкций программы (не только математических выражений).
- Сопроцессор на уровне своей системы команд поддерживает большую номенклатуру типов данных: три формата целых чисел, три формата вещественных чисел, десятичные числа. При разработке вычислительных алгоритмов и подборе для их реализации команд сопроцессора следует помнить, что сопроцессор поддерживает только один внутренний формат представления данных — вещественные числа расширенного формата. По этой причине команды сопроцессора, работающие с форматами, отличными от расширенного, вынуждены выполнять дополнительное преобразование данных. Операция преобразования требует дополнительного количества (и немалого) машинных тактов, что не может не сказаться на общем времени выполнения программы.
- Система команд сопроцессора состоит из нескольких групп, призванных удовлетворить основные потребности программиста в средствах реализации большинства вычислительных алгоритмов. При отсутствия поддержки на уровне команд сопроцессора каких-либо математических операций они довольно просто могут быть реализованы с помощью математических формул приведения через существующие команды.
- В процессе работы внутри сопроцессора могут возникать различные ситуации, требующие внешнего вмешательства. Их называют исключениями. Исключения разбиты на 6 типов, которым соответствуют по 6 битов в регистрах SWR и CWR. Эти биты позволяют управлять обработкой соответствующих исключе-

ний. Биты в SWR фиксируют возникновение исключений определенного типа. Биты в CWR определяют способ обработки возникших исключений. Если при возникновении исключения некоторого типа соответствующий этому исключению бит в CWR равен 1, это означает, что обработка исключения данного типа замаскирована и сопроцессор должен сам исправить ошибочную ситуацию. Если соответствующий возникшему исключению бит в CWR равен 0, это означает, что программист сам желает исправить ошибочную ситуацию. Для этого он должен написать обработчик исключения.

- Разработку программ удобно вести с использованием отладчика Turbo Debuger, который предоставляет полную информацию о состоянии вычислительного процесса, использующего команды процессора и сопроцессора.

Вместо заключения...

Ну вот и все! Прощаясь с читателем, позволю себе несколько мыслей о жизни вслух (не примите их в качестве наставлений).

В данной книге я попытался посмотреть на язык ассемблера как на обычный язык программирования. При этом хотелось показать, что этот язык ничем не хуже, а в чем-то даже лучше привычных читателю языков высокого уровня. Осознание этого особенно важно для начинающих программистов, которые зачастую с первых шагов в своей карьере попадают в «теплые, нежные, заботливые... (дальше придумайте сами)» объятия различных интегрированных оболочек, сред, студий... Опасно это тем, что такой молодой, неопытный, наивный программист постепенно всецело поддается нежному шелесту этих интегрированных продуктов и засыпает. Спать он может долго, может быть, даже на протяжении всей своей профессиональной карьеры, и, возможно, при этом ему будет сниться сон, суть которого в том, что:

- не нужно думать об оптимизации программы, ведь памяти и запаса быстродействия у современного процессора столько, что хватит на мгновенное псевдопараллельное выполнение десятка даже самых бездарных программ;
- не нужно думать о работе с «железом», ведь современная ОС имеет столько драйверов устройств, что вряд ли встретится ситуация, когда к компьютеру потребуется подключить что-то нестандартное;
- не нужно думать о защите своих программ — от вирусов нас защитит программа-антивирус;
- боже упаси лезть в чужой исполняемый код, даже если ты «лопаешься» от любопытства;
- и, наконец, нет необходимости что-то знать и о самом компьютере — стоит себе ящик, работает, когда его включишь, — чего еще нужно?

Обычно сон прерывается в тот момент, когда человек осознает, что сидит на совершенно конкретном компьютере, функционирующем на основе определенных программно-аппаратных средств (со всеми их изъянами и достоинствами), которые иногда требуют прямого вмешательства человека в свою работу. В этом случае можно, конечно, обратится к специалисту, но какой же ты тогда профессионал?.. Судить обо всем этом, конечно, читателю. Может быть, я в чем-то и не прав. Ведь компьютер уже давно сам стал повседневным рабочим инструментом людей множества профессий, которым действительно все сказанное раньше «до лампочки». Но если книга у вас в руках, то с большой долей вероятности можно предположить, что я не одинок в своих мыслях, и вы — человек, страдающий профессиональной бессонницей. Вас трудно ввести в заблуждение обещаниями легкого достижения поставленных целей, и вы способны самостоятельно выбрать себе дорогу для достижения вершины своей карьеры кратчайшим маршрутом.

Успехов вам на этом пути!

Приложение

Система команд процессоров IA-32

Данное приложение содержит описание системы команд процессоров архитектуры IA-32 до Pentium IV включительно. Команды разбиты на 4 группы: целочисленные команды, команды сопроцессора, команды MMX-расширения и команды XMM-расширения. В пределах каждой группы команды расположены в алфавитном порядке. Для каждой команды приведены следующие данные.

- Схема команды, поясняющая состав и назначение операндов.
- Название команды с кратким описанием ее назначения.
- Описание действия команды.
- Описание флагов после выполнения команды, при этом приводятся сведения только о флагах, изменяемых командой, и используются следующие обозначения:
  - 1 — флаг устанавливается (равен 1);
  - 0 — флаг сбрасывается (равен 0);
  - r — значение флага зависит от результата выполнения команды;
  - ? — после выполнения команды флаг не определен.
- Машинные коды для всех возможных сочетаний операндов команды. Описание машинного кода производится в шестнадцатеричном виде — каждый байт машинного представления команды воспроизводится двумя шестнадцатеричными цифрами. При описании машинного кода используются следующие обозначения:
  - /цифра — здесь цифра (от 0 до 7) представляет содержимое трехразрядного поля reg в байте mod r/m, используемое как часть кода операции;
  - /r — означает, что байт mod r/m команды содержит как регистровый операнд, так и операнд r/m;
  - cb, cw, cd, cp — одно-, двух-, четырех- или шестибайтное значение, следующее за полем кода операции и используемое для определения смещения в сегменте кода (и возможное новое значение для сегментного регистра кода);
  - ib, iw, id — одно-, двух- или четырехбайтный непосредственный операнд команды, который следует за полем кода операции, байтами mod r/m или sib, при этом код операции определяет, является ли непосредственный операнд знаковым значением, а все слова и двойные слова приводятся в порядке «младший байт по младшему адресу»;
  - +rb, +rw, +rd — код регистра (от 0 до 7), добавляемый в байт кода операции, приводимого при описании машинного кода первым значением (см. также рис. 3.1 и табл. 3.3–3.5);

- +i — число (от 0 до 7), используемое в машинном представлении команд сопроцессора, когда один из операндов является регистром ST(i) из стека регистров сопроцессора (i добавляется к одиночному байту кода операции, значение которого приведено при описании машинного кода первым).

Ж Машинные коды команды сопровождаются описанием синтаксиса команды для соответствующего сочетания операндов. При описании синтаксиса используются следующие обозначения:

П rel8 — относительный адрес из диапазона -128...+127, точка отсчета — конец данной машинной команды;

П rel16, ret32 — относительные адреса в пределах сегмента кода, содержащего данную команду, используемые для операндов с размером операнда 16 (use16) и 32 (use32) бита соответственно;

П ptr16:16 и ptr16:32 — дальние указатели (обычно на адрес в сегменте кода, отличном от текущего), используемые при установленном атрибуте размера операнда 16 битов и 32 бита соответственно (запись «16:16» или «16:32» означает, что первая часть указателя является селектором или значением сегментного регистра кода, вторая — смещением в целевом сегменте кода);

П r8, r16, r32 — операнд в одном из регистров размером байт (AL, CL, DL, BL, AH, CH, DH, BH), слово (AX, CX, DX, BX, SP, BP, SI, DI) или двойное слово (EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI);

D i8, i16, i32 — непосредственный операнд размером байт (-128...+127), слово (-32 768...+32 767) или двойное слово (-2 147 483 648...+2 147 483 647);

П m — операнд в памяти размером 16 или 32 бита;

П m8, m16, т32, m48, т64, m128 — операнд в памяти размером байт, слово, двойное слово, 48/64/128 бит;

D r/m8 — байтовый операнд, который содержится либо в одном из регистров размером один байт (AL, CL, DL, BL, AH, CH, DH, BH), либо в ячейке памяти размером один байт;

D r/m16 — операнд в регистре размером в слово (AX, CX, DX, BX, SP, BP, SI, DI) или в ячейке памяти размером в слово (используется в командах, для которых атрибут размера операнда равен 16 бит);

D r/m32 — операнд в регистре размером в слово (EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI) или в ячейке памяти размером в двойное слово (используется в командах, для которых атрибут размера операнда равен 32 битам);

П ml6:16, m16:32 — операнд в памяти, содержащий дальний указатель в виде двух чисел: число слева соответствует селектору сегмента указателя, второе число является смещением в сегменте, на который указывает этот селектор;

D m16&32, m16&16, m32&32 — пары операндов в памяти, каждый элемент которых имеет размер, указанный слева и справа от знака &;

P moffs8, moffs16, moffs32 — переменная (смещение в памяти) типа байт, слово, двойное слово, требующаяся для выполнения некоторых вариантов команды mov (байт mod r/m не используется, адрес задается простым смещением относительно базы сегмента);

П sreg — сегментный регистр, кодировка в машинном коде: ES = 0; CS = 1; SS = 2; DS - 3; FS = 4; GS = 5;

P m32fp, m64fp, m80fp — операнды в формате сопроцессора (с плавающей точкой) в памяти;

P m16int, m32int, m64int — целочисленные операнды в памяти, используемые в командах сопроцессора;

P ST или ST(0) — верхний элемент стека сопроцессора;

P ST(i) — *г-н* элемент стека сопроцессора (*i* = 0...7);

P rmmx0.. .rmmx7 — операнд в одном из регистров целочисленного расширения MMX;

P rmmx/m32 — младшая часть (32 бита) MMX-регистра или 32-разрядный операнд в памяти;

- rmmx/m64 — MMX-регистр или 64-разрядный операнд в памяти;
- rxmm0...rxmm7 — операнд в одном из регистров расширения MMX с плавающей точкой (в XMM-регистре);
- rxmm/m32 — XMM-регистр или 32-разрядный операнд в памяти;
- rxmm/m64 — XMM-регистр или 64-разрядный операнд в памяти;
- rxmm/m128 — XMM-регистр или 128-разрядный операнд в памяти.

Для экономии места при описании некоторых команд могут быть опущены одна или несколько перечисленных ранее позиций. Например, отсутствие описания действия команды говорит о том, что сведений, приведенных в названии команды с кратким описанием ее назначения, достаточно для ее практического применения. Отсутствие описания флагов означает, что данная команда не изменяет флагов.

Некоторые регистры программной модели процессора имеют внутреннюю структуру. Чтобы идентифицировать поле такого регистра, в описании команд оно воспроизводится следующим образом: имя_регистра.имя_поля.

Целочисленные команды

AAA

- AAA
- 37
- Корректировка результата сложения одноразрядных неупакованных BCD-чисел командой ADD.

Действие: если младший полубайт регистра AL > 9 или AF = 1, то: (AL) = (AL) + 6; (AH) = (AH) + 1; AF = 1, CF = 1. В противном случае AF = CF = 0. В обоих случаях (AL) = (AL) AND 0fh.

Флаги: OF = ? SF = ? ZF = ? AF = r PF = ? CF = r

AAD

- AAD
- D50A
- Подготовка двухзначного неупакованного BCD-числа в регистре AX для операции деления.

Действие: (AL) = (AH x 10) + AL; (AH) - 00h.

Замечание: процессор воспринимает другой машинный код этой команды, не имеющий мнемоники, — D5 i8. Ее действие: (AL) = (AH x i8) + AL; (AH) = 00h.

Флаги: OF = ? SF = r ZF = r AF = ? PF = r CF = ?

AAM

- AAM
- D40A
- Коррекция результата умножения двух неупакованных BCD-чисел.

Действие: разделить содержимое регистра AL на 10; частное записать в регистр AH, остаток — в регистр AL.

Замечание: процессор воспринимает другой машинный код этой команды — D4 i8 (без мнемоники). Ее действие: разделить содержимое регистра AL на i8; частное записать в регистр AH, остаток — в регистр AL. Если i8 = 0, то процессор генерирует исключение ошибки деления #DE.

Флаги: OF = ? SF = r ZF = r AF = ? PF = r CF - ?

AAS

- AAS
- 3F

- Коррекция результата вычитания командой SUB двух неупакованных одноразрядных BCD-чисел.

Действие: если младший полубайт регистра AL > 9 или флаг AF = 1, то значение младшего полубайта AL уменьшается на 6, значение AH уменьшается на 1, флаги AF и CF устанавливаются в 1. Если ни одно из этих условий не выполняется, команда устанавливает флаги AF = CF = 0. В обоих случаях значение старшего полубайта регистра AL обнуляется.

Флаги: OF = ? SF = ? ZF = ? AF = r PF = ? CF = r

ADC

- ADC приемник, источник

14 ib	ADC AL,imm8
15 iw	ADC AX,imm16
15 id	ADC EAX,imm32
80 /2 ib	ADC r/m8,imm8
81 /2 iw	ADC r/m16,imm16
81 /2 id	ADC r/m32,imm32
83 /2 ib	ADC r/m16,imm8
83 /2 ib	ADC r/m32,imm8
10 /r	ADC r/m8,r8
11 /r	ADC r/m16,r16
11 /r	ADC r/m32,r32
12 /r	ADC r8,r/m8
13 /r	ADC r16,r/m16
13 /r	ADC r32,r/m32

- Сложение с учетом значения флага переноса CF.

Действие: приемник = приемник + источник + CF.
Флаги: OF = r SF = r ZF = r AF - r PF = r CF = r

ADD

- ADD приемник, источник

04 ib	ADD AL,imm8
05 iw	ADD AX,imml6
05 id	ADD EAX,imm32
80 /0 ib	ADD r/m8,imm8
81 /0 iw	ADD r/m16,imm16
81 /0 id	ADD r/m32,imm32
83 /0 ib	ADD r/m16,imm8
83 /0 ib	ADD r/m32,imm8
00 /r	ADD r/m8,r8
01 /r	ADD r/m16,r16
01 /r	ADD r/m32,r32
02 /r	ADD r8,r/m8
03 /r	ADD r16,r/m16
03 /r	ADD r32,r/m32

- Сложение двух целочисленных двоичных операндов.

Действие: приемник = приемник + источник.
При переполнении приемника флаг CF устанавливается в 1.
Флаги: OF = r SF = r ZF = r AF = r PF = r CF = r

AND

- AND приемник, источник

24 ib	AND AL,imm8
25 iw	AND AX,imm16

25 id AND EAX,imm32
80 /4 ib AND r/m8,imm8
81 /4 iw AND r/m16,imm16
81 /4 id AND r/m32,imm32
83 /4 ib AND r/m16,imm8
83 /4 ib AND r/m32,imm8
20 /r AND r/m8,r8
21 /r AND r/m16,r16
21 /r AND r/m32,r32
22 /r AND r8,r/m8
23 /r AND r16,r/m16
23 /r AND r32,r/m32

- Логическое И.

Действие: приемник = приемник AND источник.
Флаги: OF = 0 SF = r ZF = r AF = ? PF = r CF = 0

ARPL

- ARPL приемник, источник
- 63 /r ARPL r/m16,r16
- Настройка поля RPL селектора.

Действие: операнд приемник содержит селектор вызываемой программы, операнд источник — селектор вызывающей программы. Команда ARPL сравнивает биты RPL селектора приемника с битами RPL селектора источника и в зависимости от результатов этого сравнения выполняет действия:

- если RPL приемника < RPL источника, то ZF = 1 и RPL приемника = RPL источника;
- если RPL приемника > RPL источника, то: ZF = 0.

Флаги: ZF = r

BOUND

- BOUND индекс, границы_массива
- 62 /r BOUND r16,m16&16
  62 /r BOUND r32,m32&32
- Контроль нахождения индекса массива в границах.

Действие: сравнить значение в 16/32-разрядном регистре индекс с диапазоном значений, первое из которых — это нижний индекс, второе — верхний индекс плюс размер операнда в байтах. Значения нижнего и верхнего индексов расположены последовательно в двух ячейках памяти размером слово/двойное слово, адресуемых операндом границы_массива. Если в результате проверки значение из регистра вышло за пределы указанного диапазона значений, то возбуждается прерывание с номером 5, если нет, программа продолжает выполнение.

BSF

- BSF результат, источник
- OF BC BSF r16,r/m16
  OFBC BSF r32,r/m32
- Определение номера позиции в операнде источник крайнего справа единичного бита.

Действие: просмотр битов операнда источник, начиная с младшего. Номер позиции первого единичного бита слева записывается в регистр результат, флаг ZF устанавливается в 0. Если единичных битов нет, то флаг ZF устанавливается в 1, а регистр результат оказывается неопределенным.

Флаги: OF = ? SF - ? ZF = r AF = ? PF = ? CF = ?

BSR

- BSR результат, источник
- 0F BD BSR r16,r/m16
  0F BD BSR r32,r/m32
- Определение номера позиции крайнего слева единичного бита в операнде источник.

Действие: команда просматривает биты операнда источник начиная со старшего бита 15/31, и, если встречается единичный бит, флаг ZF устанавливается в 0, а в регистр результат записывается номер позиции (отсчет осуществляется относительно нулевой позиции), в которой встретился единичный бит. Если единичных битов нет, то флаг ZF устанавливается в 1. Диапазон значений результата зависит от разрядности второго операнда: для 16/32-разрядных операндов это, соответственно, 0...15/0...31.

Флаги: OF = ? SF = ? ZF = r AF = ? PF = ? CF = ?

BSWAP

- BSWAP источник
- 0F C8 + rd BSWAP r32
- Изменение порядка следования байтов в операнде источник.

Действие:

- TEMP < источник;
- источник(7...0) < TEMP(31...24);
- источник(15...8) < TEMP(23...16);
- источник(23...16)< TEMP(15...8);
- источник (31...24) < TEMP(7...0).

Здесь, TEMP — временная ячейка памяти (32 бита), которую процессор выделяет в своих внутренних структурах для выполнения этой операции.

BT

- BT источник, индекс
- 0F A3 BT r/m16,r16
  0F A3 BT r/m32,r32
  0F BA /4 ib BT r/m16,imm8
  0F BA /4 ib BT r/m32,imm8
- Определение значения конкретного бита в операнде источник.

Действие: номер проверяемого бита операнда источник задается операндом индекс. После выполнения команды флаг CF устанавливается значением проверяемого бита.

Флаги: OF = ? SF = ? ZF = ? AF = ? PF = ? CF = r

BTC

- BTC источник, индекс
- 0F BB BTC r/m16,r16
  0F BB BTC r/m32,r32
  0F BA /7 ib BTC r/m16,imm8
  0F BA /7 ib BTC r/m32,imm8
- Определение и инвертирование значения заданного бита в операнде источник.

Действие: номер проверяемого бита операнда источник задается операндом индекс (значение из диапазона 0...31). После выполнения команды значение выбранного бита инвертируется, а флаг CF устанавливается исходным значением этого бита.

Флаги: OF = ? SF = ? ZF = ? AF = ? PF = ? CF = r

BTR

- BTR источник, индекс
- 0F B3 BTR r/m16,r16
  0F B3 BTR r/m32,r32
  0F BA /6 ib BTR r/m16,imm8
  0F BA /6 ib BTR r/m32,imm8
- Определение значения заданного бита в операнде источник и сброс его в 0.

Действие: номер проверяемого бита операнда источник задается операндом индекс (значение из диапазона 0...31). После выполнения команды флаг CF устанавливается исходным значением этого бита, а сам бит устанавливается в 0.

Флаги: OF = ? SF = ? ZF = ? AF = ? PF = ? CF = r

BTS

- BTS источник, индекс
- 0F AB BTS r/m16,r16
  0F AB BTS r/m32,r32
  0F BA /5 ib BTS r/m16,imm8
  0F BA /5 ib BTS r/m32,imm8
- Определение значения заданного бита в операнде источник и установка его в 1.

Действие: номер проверяемого бита задается операндом индекс (значение из диапазона 0...31). После выполнения команды флаг CF устанавливается исходным значением этого бита, а сам бит устанавливается в 1.

Флаги: OF = ? SF = ? ZF = ? AF = ? PF = ? CF = r

CALL

- CALL цель
- E8 cw CALL rel16
  E8 cd CALL rel32
  FF /2 CALL r/m16
  FF /2 CALL r/m32
  9A cd CALL ptr16:16
  9A cp CALL ptr16:32
  FF/3 CALL m16:16
  FF/3 CALL m16:32
- Вызов процедуры или переключение задачи.

Действие: далее перечислены возможные варианты задания операнда цель.

- rel16/32 — близкий относительный переход. Значение rel16/32 трактуется как знаковое и является смещением перехода относительно следующей за CALL команды в сегменте кода, то есть адрес цели равняется EIP/IP + rel16/32. В стек заносится содержимое EIP/IP.
- r16/32 или m16/32 — близкий абсолютный косвенный переход. Цель — регистр r16/32 или m16/32. Содержимое этого регистра является смещением команды, которой передается управление, в текущем сегменте кода. В EIP/IP загружается содержимое из r16/32 или m16/32. В стек заносится содержимое EIP/IP.
- m16:16(32) — дальний абсолютный косвенный переход. Цель — адрес ячейки памяти размером 32/48 бит со структурой m16:16(32), содержащей компоненты адреса перехода.
- ptr16:16(32) — дальний абсолютный переход. Цель — компоненты полного адреса в виде 4- или 6-разрядного указателя, по которому необходимо произвести переход.

Особенности выполнения команды CALL в двух последних вариантах определяются режимом работы процессора.

- В реальном режиме или режиме виртуального процессора 8086 (PE = 0 или PE = 1 и VM = 1) в EIP/IP и CS загружаются значения смещения и адреса сегмента из указателя в памяти или команды CALL. В стек заносится содержимое EIP/IP и CS.
- В защищенном режиме (PE = 1 и VM = 0) анализируется байт прав доступа AR в дескрипторе целевого сегмента. В зависимости от его значения производится пять типов перехода к цели:
  - передача управления подчиненному сегменту кода;
  - передача управления неподчиненному сегменту кода;
  - передача управления через шлюз вызова;
  - переключение задачи (селектор соответствует дескриптору — шлюзу задачи);
  - переключение задачи (селектор соответствует дескриптору — сегменту TSS).

Флаги (кроме случая переключения задачи): не изменяются.

При переключении задачи значения флагов изменяются содержимым регистра EFLAGS в сегменте состояния TSS задачи, на которую производится переключение.

CBW/CWDE

- CBW/CWDE
- 98
- CBW — преобразование байта в слово,
  CWDE — преобразование слова в двойное слово.

Действие: команда копирует знаковый бит регистра AL(AX) на все биты регистра AH (EAX).

CWD/CDQ

- CWD/CDQ
- 99
- CWD — преобразование слова в двойное слово,
  CDQ — преобразование двойного слова в учетверенное слово.

Действие: команда CWD копирует значение старшего бита регистра AX на все биты регистра DX; команда CDQ копирует знаковый бит регистра EAX на все биты регистра EDX.

CLC

- CLC
- F8
- Сброс флага переноса CF.

Флаги: CF = 0

CLD

- CLD
- FC
- Сброс флага направления DF.

Флаги: OF = 0

CLFLUSH

- CLFLUSH операнд
- 0F AE /7
- Аннулирование строки кэша, содержащей операнд.

Действие: аннулирует строки кэш-памяти, которая содержит линейные адреса, определяемые операндом, для кэшей всех уровней (данных и команд). Если аннулируемая строка помечена как измененная (dirty), то предварительно она записывается на диск.

CLI

- CLI
- FA
- Сброс флага прерывания IF.

Флаги: IF = 0 (если CPL< = IOPL)

Замечания: флаг IF не сбрасывается, если CR0.PE - 1, CPL > IOPL и VM = 0 или CR0.PE = 1, IOPL < 3 и VM = 1. Флаг IF и команды CLI и STI не влияют на генерацию исключений и прерывания NMI.

CLTS

- CLTS
- 0F 06
- Сброс флага переключения задач TS в регистре CR0 (бит 3) в 0.

CMC

- CMC
- F5
- Инвертирование флага переноса CF.

Флаги: CF = r

CMOVcc

- CMOVcc приемник, источник
- Передача данных при выполнении условий, определяемых состоянием соответствующих флагов.

Действие: если состояние флагов (см. табл. П.1) соответствует условиям выполнения команды, то производится пересылка значения источника в приемник. В противном случае выполнение команды завершается без пересылки.

Таблица П.1. Состояние флагов при выполнении команды CMOVcc

Машинный код	Мнемокод	Флаги	Пересылка из источника в приемник
0F40 cw/cd	CMOVO r16,r/m16 (r32,r/m32)	OF = 1	Если переполнение
0F 41 cw/cd	CMOVNO r16,r/m16 (r32,r/m32)	OF = 0	Если нет переполнения
0F42 cw/cd	CMOVB r16,r/m16 (r32,r/m32) CMOVNAE r16,r/m16 (r32,r/m32)	CF = 1	Если ниже (не выше или равно)
0F42 cw/cd	CMOVC r16,r/m16 (r32,r/m32)	CF = 1	Если перенос
0F43 cw/cd	CMOVNC r16, r/m16 (r32,r/m32)	CF = 0	Если нет переноса
0F44 cw/cd	CMOVE r16,r/m16 (r32,r/m32) CMOVZ r16,r/m16 (r32,r/m32)	ZF = 1	Если равно (нуль)
0F45 cw/cd	CMOVNE r16, r/m16 (r32,r/m32) CMOVNZ r16, r/m16 (r32,r/m32)	ZF = 0	Если не равно (не нуль)
0F46 cw/cd	CMOVBE r16, r/m16 (r32,r/m32) MOVNA r16, r/m16 (r32,r/m32)	CF = 1| ZF = 1	Если ниже или равно (не выше)
0F47 cw/cd	CMOVA r16, r/m16 (r32,r/m32) MOVNBE r16, r/m16 (r32,r/m32)	CF = 0& ZF = 0	Если выше (не ниже или равно)

продолжение ⇨

Таблица П.1 *(продолжение)*

Машинный код	Мнемокод	Флаги	Пересылка из источника в приемник
OF48cw/cd	CMOVS r16, r/m16 (r32,r/m32)	SF = 1	Если знак
OF49cw/cd	CMOVNS r16, r/m16 (r32,r/m32)	SF = 0	Если не знак
OF4Acw/cd	CMOVP r16, r/m16 (r32,r/m32) MOVPE r16, r/m16 (r32,r/m32)	PF = 1	Если паритет (четное количество единичных битов)
OF4Bcw/cd	CMOVNP r16, r/m16 (r32,r/m32) CMOVPO r16, r/m16 (r32,r/m32)	PF = 0	Если не паритет
OF 4C cw/cd	CMOVL r16,r/m16 (r32,r/m32) CMOVNGE r16,r/m16 (r32,r/m32)	SF<>OF	Если меньше (не больше или равно)
OF4Dcw/cd	CMOVGE rl6, r/m16 (r32,r/m32) CMOVNL rl6, r/ml6 (r32,r/m32)	SF = OF	Если больше или равно (не меньше)
OF4Ecw/cd	CMOVLE r16, r/ml6 (r32,r/m32) CMOVNG r16, r/m16 (r32,r/m32)	ZF = 1| SFoOF	Если меньше или равно (не больше)
OF4Fcw/cd	CMOVG r16, r/m16 (r32,r/m32) CMOVNLE r16,r/m16 (r32,r/m32)	ZF = 0& SF = OF	Если больше (не меньше или равно)

CMP

- CMP операнд_1, операнд_2
- 3Cib CMP AL,imm8
  3D iw CMP AX, imm16
  3D id CMP EAX,imm32
  80 /7 ib CMP r/m8,imm8
  81 /7 iw CMP r/m16,imm16
  81 /7 id CMP r/m32,imm32
  83 /7 ib CMP r/ml6,imm8
  83 /7 ib CMP r/m32,imm8
  38 /r CMP r/m8,r8
  39 /r CMP r/m16,r16
  39 /r CMP r/m32,r32
  3A /r CMP r8,r/m8
  3B/r CMP r16,r/m16
  3B/r CMP r32,r/m32
- Сравнение двух операндов.

Действие: операнды операнд_1 и операнд_2 сравниваются методом вычитания, при этом сами операнды не изменяются. По результатам сравнения устанавливаются флаги (см. описание команды SBB).

Флаги: OF = r SF = r ZF = r AF = r PF = r CF = r

CMPS/CMPSB/CMPSW/CMPSD

- CMPS приемник, источник
  CMPSB/CMPSW/CMPSD
- A6 CMPS DS:(E)SI,ES:(E)DI
  A7 CMPS DS:SI, ES:DI
  A7 CMPS DS:ESI, ES:EDI
  A6 CMPSB
  A7 CMPSW
  A7 CMPSD

- Сравнение цепочек байтов/слов/двойных слов. Адреса сравниваемых элементов цепочек предварительно загружаются: адрес источника в пару регистров DS:ESI/SI; адрес приемника в пару регистров ES:EDI/DI.

Действия:

1. Вычесть элементы (источник - приемник).
2. В зависимости от состояния флага DF изменить значение регистров ESI/SI и EDI/DI:
   - если DF = 0 — содержимое регистров увеличивается на длину элемента цепочки;
   - если OF = 1 — содержимое регистров уменьшается на длину элемента цепочки.
3. В зависимости от результата вычитания устанавливаются флаги:
   - если элементы цепочек не равны, то CF = 1, ZF = 0;
   - если элементы цепочек или цепочки в целом равны, то CF = O, ZF = 1.
4. При наличии префикса повторения выполнить определяемые им действия (см. описание команд REPE/REPNE).

Флаги: OF = r SF = r ZF = r AF = r PF = r CF = r

CMPXCHG

- CMPXCHG приемник, источник
- OF BO/r CMPXCHG r/m8,r8
  OF Bl/r CMPXCHG r/m16,r16
  OF Bl/r CMPXCHG r/m32,r32
- Сравнение с аккумулятором и обмен.

Действие: если аккумулятор (AL/AX/EAX) и приемник не равны, то установить ZF в 0 и переслать содержимое приемника в аккумулятор (AL/AX/EAX). Если аккумулятор и приемник равны, то установить ZF в 1 и переслать источник в приемник.

Флаги: OF = r SF = r ZF = r AF = r PF = r CF = r

CMPXCHG8B

- CMPXCHG8B приемник
- OF C7 /1 m64
- Сравнение и обмен восьми байтов.

Действия: сравнить содержимое регистров EDX:EAX и ячейки памяти приемник (m64). Если (EDX:EAX) = (т64), то ZF = 1, (т64) - (ECX:EBX). В противном случае: ZF =0, (EDX:EAX) = (т64).

Флаги: ZF = r

CPUID

- CPUID
- OF A2
- Получение информации о текущем процессоре.

Действие: для получения информации о процессоре необходимо в регистр EAX поместить параметр — одно из значений O, 1 или 2.

Если EAX = 0, то в регистрах EAX, EBX, EDX, ECX формируется следующая информация:

- EAX = п, где n — максимально допустимое значение параметра, которое может быть помещено в регистр EAX для задания режима сбора информации;
- EBX + EDX + ECX — в этих регистрах содержится строка-идентификатор процессора GenuineIntel (*genuine* — подлинный, истинный):
  - EBX = 756E6547h "Genu" (G in BL);
  - ECX = 6C65746Eh "ntel" (n in CL);

- EDX - 49656E69h "ineI" (i in DL).

Если EAX = 1, то в регистрах процессора сформируется следующая информация:

- EAX = n — информация о процессоре (см. табл. П.2 и П.3);
- EDX = п — информация о возможностях процессора (см. табл. П.4).

Если EAX = 2, то в регистрах EAX, EBX, ECX и EDX формируется информация о кэш-памяти первого уровня и TLB-буферах. Первый байт регистра EAX содержит число, означающее, сколько раз необходимо последовательно выполнить команду CPUID для получения полной информации о кэш-памяти первого уровня и TLB-буферах. Другие байты регистра EAX и все байты регистров EBX, ECX и EDX содержат однобайтовые дескрипторы, характеризующие кэш-память и TLB-буферы (см. документацию по процессору). Старший бит каждого регистра характеризует достоверность информации в регистре. Если он равен нулю, то информация достоверна, иначе — регистр не используется.

Таблица П.2. Поля регистра EAX после выполнения команды CPUID (при EAX = 1)

Биты EAX	Назначение
0...3	Версия изменений модели (stepping ID)
4...7	Модель в семействе (см. табл. А.3)
8...11	Семейство процессоров (см. табл. А.3)
12...13	Тип процессора (00 — обычный процессор; 01 — Overdrive-процессор; 10 — процессор для использования в двухпроцессорных системах)

Таблица П.3. Значения битов 4...7 и 8...11 регистра EAX

Биты EAX (8... 11)	Биты EAX (4...7)	Тип процессора
0100	0000 или 0001	I486DX
0100	0010	I486SX
0101	0010	Pentium 75-200
0101	0100	Pentium MMX 166-200
0110	0001	Pentium Pro
0110	OOH	Pentium II, модель 3
0110	0101	Pentium II, модель 5, Pentium II Xeon
0110	0110	Celeron, модель 6
0110	0111	Pentium III и Pentium III Xeon
0110	OOП	Pentium II OverDrive
1111	0000	Pentium IV

Таблица П.4. Поля регистра EDX после выполнения команды CPUID (при EAX = 1)

Биты EDX	Назначение (если биты установлены)
0	Присутствует сопроцессор с набором команд i387
1	Поддержка расширенных возможностей обработки прерываний в режиме виртуального процессора i8086
2	Процессор поддерживает точки прерывания ввода-вывода (точки останова по обращению к портам) для предоставления расширенных возможностей отладки и доступ к регистрам DR4 и DR5. Флаг CR4.DE - 1
3	Процессор поддерживает 4-мегабайтные страницы (бит CR4.PSE)

Биты EDX	Назначение (если биты установлены)
4	Поддержка счетчика меток реального времени TSC (команда RDTSC)
5	Поддержка команд RDMSR и WRMSR для работы с модельно-зависимыми регистрами
6	Процессор поддерживает физические адреса большие, чем 32 бита (CR4.PAE), расширенный формат элемента таблицы страниц, дополнительный уровень трансляции страничного адреса и 2-мегабайтные страницы
7	Поддержка исключения машинного контроля 18 (Machine Check Exception, MCE)
8	Поддержка инструкции CMPXCHG8B
9	Процессор содержит программно-доступный контроллер прерываний APIC, который доступен для использования
10	Резерв
И	Поддержка инструкций быстрых системных вызовов SYSENTER и SYSEXIT
12	Поддержка регистра управления кэшированием MTRR_CAP (относится к MSR-регистрам)
13	Поддержка работы с битом G, определяющим глобальность страницы в PTDE и PTE. Бит CR4.PGE = 1
14	Поддержка архитектуры машинного контроля (MSR-регистр MCG_CAP)
15	Поддержка команд CMOVcc, FMOVcc и FCOMI, если установлен бит EDX.O = 1 (см. выше)
16	Поддержка таблицы физических атрибутов страниц PAT
17	Поддержка 36-разрядной физической адресации с 4-мегабайтными страницами
18	Процессор поддерживает собственную идентификацию по уникальному 96-разрядному номеру PPN (Physical Processor Number), и эта поддержка активна
19	Поддержка команды CLFLUSH
20	Резерв
21	Процессор поддерживает способность записывать информацию в резидентный буфер памяти
22	Поддержка внутренних MSR-регистров для мониторинга температуры процессора и программного регулирования производительности процессора (ACPI)
23	Поддержка целочисленного MMX-расширения
24	Процессор поддерживает команды FXSAVE и FXRSTOR
25	Поддержка MMX-расширения (SSE) с плавающей точкой
26	Поддержка MMX-расширения (SSE2) с плавающей точкой
27	Поддержка самослежения (self snoop) за состоянием кэша
28	Процессор поддерживает технологию Hyper Threading
29	Процессор обеспечивает схемотехнический термальный контроль
30	Процессор архитектуры Intel Itanium; поддерживается соответствующий набор команд
31	Поддержка контакта FERR#/PBE# для сигнализации о задержанном прерывании

Перед своей работой команда CPUID выполняет сериализацию команд записи в память.

DAA

- DAA
- 27

- Десятичная коррекция результата сложения двух упакованных BCD-чисел сцелью получения правильного двузначного десятичного числа.

Действия:

1. Если AF = 1 или значение младшей тетрады AL > 9, то: (AL) = (AL) + 6; AF = 1; при возникновении переноса при сложении установить флаг CF. Иначе — AF = 0.
2. Если CF = 1 или значение старшей тетрады AL > 9, то: (AL) = (AL) + 60; CF = 1. Иначе — CF = 0.

Флаги: OF = ? SF = r ZF = r AF = r PF = r CF = r

DAS

- DAS
- 2F
- Десятичная коррекция после вычитания двух BCD-чисел в упакованном формате.

Действия:

1. Если AF = 1 или значение младшей тетрады AL > 9, то (AL) = (AL) - 6; AF = 1; в случае заема при вычитании установить флаг CF. Иначе — AF = 0.
2. Если CF = 1 или значение старшей тетрады AL > 9, то (AL) = (AL) -60; CF = 1. Иначе — CF = 0.

Флаги: OF = ? SF = r ZF = r AF = r PF = r CF = r

DEC

- DEC операнд
- FE /1 DEC r/m8
  FF/1 DEC r/m16
  FF /1 DEC r/m32
  48 + rw DEC r16
  48 + rd DEC r32
- Уменьшение значения операнда на единицу.

Флаги: OF = r SF - r ZF = r AF = r PF = r, флаг CF не меняется.

DIV

- DIV делитель
- F6 /6 DIV r/m8
  F7 /6 DIV r/m16
  F7 /6 DIV r/m32
- Беззнаковое деление.

Действие: делимое задается неявно, и его размер зависит от размера делителя, который явно указывается в команде. Местоположения делимого, делителя, частного и остатка — в зависимости от их размерности (табл. П.5).

Флаги: OF = ? SF = ? ZF = ? AF = ? PF = ? CF = ?

Таблица П.5. Местоположения делимого, делителя, частного и остатка после выполнения команды DIV

Размер операнда	Делимое	Делитель	Частное	Остаток	Максимальное частное
Слово (байт)	AX	r/m8	AL	AH	255
Двойное слово (слово)	DX:AX	r/m16	AX	DX	65 535
Учетверенное слово (двойное слово)	EDX:EAX	r/m32	EAX	EDX	$2^{32} - 1$

ENTER

- ENTER размер_кадра, лексический_уровень
- C8 iw 00 ENTER imm16,0
  C8 iw 01 ENTER imm16,1
  C8 iw ib ENTER imm16,imm8
- Установить кадр стека для локальных переменных процедуры.

Действие: операнд размер_кадра определяет размер кадра стека, операнд лексический_уровень задает лексическую вложенность кадра стека. Значение лексического уровня определяет количество указателей кадра стека, копируемых в область дисплея нового кадра стека из предыдущего кадра. Оба операнда непосредственные.

HLT

- HLT
- F4
- Останов процессора.

Действие: команда переводит процессор в состояние останова. Выполнение будет продолжено но приходу разрешенного прерывания, NMI или аппаратного сброса. Команда HLT является привилегированной.

IDIV

- IDIV делитель
- F6 /7 IDIV r/m8
  F7 /7 IDIV r/m16
  F7 /7 IDIV r/m32
- Целочисленное деление со знаком.

Действие: делимое задается неявно, и его размер зависит от размера делителя, который явно указывается в команде. Местоположения делимого, делителя, частного и остатка — в зависимости от их размера (табл. П.6).

Таблица П.6. Местоположения делимого, делителя, частного и остатка после выполнения команды IDIV

Размер операнда	Делимое	Делитель	Частное	Остаток	Максимальное частное
Слово (байт)	AX	r/m8	AL	AH	–128... +127
Двойное слово (слово)	DX:AX	r/m16	AX	DX	-32 768... +32 767
Учетверенное слово (двойное слово)	EDX:EAX	r/m32	EAX	EDX	$-2^{31}...2^{32}-1$

Остаток всегда имеет знак делимого. Знак частного зависит от состояния знаковых битов (старших разрядов) делимого и делителя.

Флаги: OF = ? SF = ? ZF = ? AF = ? PF = ? CF = ?

IMUL

- IMUL множитель_1
  IMUL множитель_1, множитель_2
  IMUL произведение, множитель_1, множитель_2
- F6 /5 IMUL r/m8
  F7 /5 IMUL r/m16
  F7 /5 IMUL r/m32

0F AF /r IMUL r16,r/m16
0F AF /r IMUL r32,r/m32
6B /r ib IMUL r16,r/m16,imm8
6B /r ib IMUL r32,r/m32,imm8
6B /r ib IMUL r16,imm8
6B /r ib IMUL r32,imm8
69 /r iw IMUL r16,r/m16,imm16
69 /r id IMUL r32,r/m32,imm32
69 /r iw IMUL r16,imm16
69 /r id IMUL r32,imm32

- Целочисленное умножение со знаком.

Действие: команда имеет три формы, различающиеся количеством операндов.

- С одним операндом — требует явного указания местоположения только одного сомножителя, который может быть расположен в ячейке памяти или регистре. Местоположение произведения зависит от размерности множителей (табл. П.7).
- С двумя операндами — первый операнд определяет местоположение первого сомножителя. На его место впоследствии будет записан результат. Второй операнд определяет местоположение второго сомножителя.
- С тремя операндами — первый операнд определяет местоположение результата, второй операнд — местоположение первого сомножителя, третий операнд может быть непосредственно заданным значением размером в байт, слово или двойное слово.

Таблица П.7. Местоположения произведения после выполнения команды IMUL

Размер множителей	Множитель 1	Множитель 2	Произведение
Байт	AL	r/m8	AX
Слово	AX	r/m16	DX:AX
Двойное слово	DX:AX	r/m32	EDX:EAX

Флаги (для однооперандной команды): OF = 1 CF = 1 — значимые биты переносятся в верхнюю половину результата; OF = 0 CF = 0 — результат помещается точно в младшей половине результата. Состояние остальных флагов: SF - ? ZF = ? AF = ? PF = ?

Флаги (для двух- и трехоперандной команды): OF = 1 CF = 1 — результат слишком большой и усекается; OF = 1 CF - 1 — размер результата точно соответствует операнду назначения. Состояние остальных флагов: SF = ? ZF - ? AF = ? PF = ?

IN

- IN аккумулятор, номер_порта

E4 ib IN AL,imm8
E5 ib IN AX,imm8
E5 ib IN EAX,imm8
EC IN AL,DX
ED IN AX,DX
ED IN EAX,DX

- Ввод операнда размером байт, слово, двойное слово из порта.

Действие: ввод операнда размером байт, слово, двойное слово из порта ввода-вывода в один из регистров AL/AX/EAX. Номер порта задается вторым операндом в виде непосредственного значения (0...255) или значения в регистре DX.

INC

- INC операнд

- FE /0 INC r/m8
  FF /0 INC r/m16
  FF /0 INC r/m32
  40 + rw INC r16
  40+ rd INC r32
- Увеличение операнда размером байт, слово, двойное слово на 1. Команда не воздействует на флаг CF.

  Флаги: OF = r SF = r ZF = r AF = r PF = r

INS/INSB/INSW/INSD

- INS приемник, порт
  INSB/INSW/INSD
- 6C INS ES:(E)DI, DX
  6D INS ES:DI, DX
  6D INS ES:EDI, DX
  6C INSB
  6D INSW
  6D INSD
- Ввод строк байтов/слов/двойных слов из порта ввода-вывода в память.

Действие: номер порта ввода-вывода хранится в регистре DX, адрес ячейки памяти — в ES: EDI/DI. Замена сегментного регистра недопустима. Команда передает элемент из порта ввода-вывода в память и в зависимости от состояния флага DF изменяет значение в регистре EDI/DI:

- если DF = 0, увеличить содержимое этих регистров на длину структурного элемента последовательности;
- если DF = 1, уменьшить содержимое этих регистров на длину структурного элемента последовательности.

При наличии префикса выполняются определяемые им действия (см. описание команды REP).

INT/INTO/INT 3

- INT номер_прерывания
- CDib INT номер_прерывания
- Вызов подпрограммы обслуживания прерывания.

- INTO
- CE INTO
- Прерывание, если переполнение.

- INT 3
- CC INT3
- Вызов подпрограммы обслуживания прерывания 3.

Действие: команда INT генерирует вызов подпрограммы обслуживания прерывания с номером (0...255), заданным операндом команды.

В реальном режиме команда INT n записывает в стек регистр флагов EFLAGS/FLAGS и адрес возврата — содержимое регистров CS и EIP/IP. Далее сбрасываются в ноль флаги IF, TF и AC, после чего управление передается программе обработки прерывания с номером n.

В защищенном режиме проверяются условия VM = 1 и IOPL<3. Если они выполняются, то возбуждается исключение #GP(0). Иначе, проверяется тип дескриптора: он должен быть шлюзом ловушки, задачи, прерывания.

- Шлюз задачи — селектор в дескрипторе шлюза указывает на дескриптор TSS в GDT. Производится переключение задач (с вложением) и отслеживаются условия возникновения исключений.

- Шлюз ловушки или прерывания — при этом возможны два типа передачи управления обработчику прерывания: прерывание в режиме виртуального процессора 8086, передача управления между уровнями привилегий.

Команды вызова обработчиков прерываний INTO и INT 3 являются специализированными: INTO инициирует прерывание с номером 4, если установлен флаг OF; INT 3 генерирует специальный однобайтовый код операции (Occh), который предназначается для вызова обработчика исключения отладки.

Флаги: в зависимости от режима работы процессора флаги IF, TF, NT, AC, RF и VM могут быть очищены. Если прерывание использует шлюз задачи, то любые флаги могут быть установлены или очищены в соответствии с образом EFLAGS в TSS новой задачи.

INVD

- INVD
- 0F 08
- Недостоверность кэш-памяти всех уровней.

Действие: очистка кэш-памяти первого уровня (внутренней) и генерация сигнала на очистку кэш-памяти второго уровня (внешней).

INVLPG

- INVLPG адрес
- 0F 01/7
- Недостоверность элемента буфера TLB.

Действие: просмотр элементов буфера TLB и выяснение, соответствует ли адрес, указанный в команде, одному из элементов этого буфера. Если соответствие выявлено, то данный элемент буфера TLB помечается как недостоверный и работа заканчивается. Если соответствия не выявлено, то работа команды заканчивается.

IRET/IRETD

- IRET/IRETD
- CF
- Возврат из прерывания

Действия: зависят от режима работы микропроцесссора.

- В реальном режиме выполняется последовательное извлечение из стека содержимого регистров EIP/IP, CS и EFLAGS/FLAGS, и работа прерванной программы возобновляется.
- В защищенном режиме действия команды IRET определяются флагами NT и VM в регистре EFLAGS, а также значением флага VM в образе EFLAGS, сохраненного в текущем стеке. В зависимости от их состояния процессор выполняет следующие виды возврата: возврат из режима V86; возврат в режим V86; возврат к коду на другом уровне привилегий; возврат из вложенной задачи. Если NT = 0, то производятся действия по возврату управления прерванной программе, при этом характер этих действий зависит от соотношения уровней привилегированности прерванной программы и программы обработки прерывания. Если NT = 1, то производятся действия по переключению задач.

Флаги: Все флаги в регистре EFLAGS могут быть модифицированы.

Jcc

- Jcc метка
- Переход, если выполнено условие cc.

Действие: команды условного перехода, в зависимости от своей мнемоники, анализируют флаги, и если проверяемое условие истинно, то производится переход к ячейке, обозначенной операндом. Если проверяемое условие ложно, то производится переход к следующей команде.

Идентификатор метка преобразуется ассемблером в непосредственные значения rel8 (use16) и rel16/rel32 (use32), которые во время выполнения добавляются к текущему значению IP/EIP. Мнемоника команд условного перехода показана в табл. П.8 (логические условия «больше» и «меньше» относятся к сравнениям целочисленных значений со знаком, а «выше» и «ниже» — к сравнениям целочисленных значений без знака).

Таблица П.8. Мнемоника команд условного перехода

Машинный код	Мнемокод	Флаги	Переход
70 cb	JO rel8	OF = 1	Короткий, если переполнение
71 cb	JNO rel8	OF = 0	Короткий, если нет переполнения
72 cb	JB rel8 JC rel8 JNAE rel8	CF = 1	Короткий, если ниже Короткий, если перепое Короткий, если не выше или равно
73 cb	JAE rel8 JNB rel8 JNC rel8	CF = 0	Короткий, если выше или равно Короткий, если не ниже Короткий, если не перенос
74 cb	JE rel8 JZ rel8	ZF = 1	Короткий, если равно Короткий, если нуль
75 cb	JNZ rel8 JNE rel8	ZF = 0	Короткий, если не нуль Короткий, если не равно
76 cb	JBE rel8 JNA rel8	CF = 1|ZF = 1	Короткий, если ниже или равно (не выше)
77 cb	JA rel8 JNBE rel8	CF = 0&ZF = 0	Короткий, если выше (не ниже или равно)
78 cb	JS rel8	SF = 1	Короткий, если знак
79 cb	JNS rel8	SF = 0	Короткий, если не знак
7A cb	JP rel8 JPE rel8	PF = 1	Короткий, если паритет (четное количество единичных битов)
7B cb	JPO rel8 JNP rel8	PF = 0	Короткий, если не паритет (нечетное количество единичных битов)
7C cb	JL rel8 JNGE rel8	SF<>OF	Короткий, если меньше (не больше или равно)
7D cb	JGE rel8 JNL rel8	SF = OF	Короткий, если больше или равно (не меньше)
7E cb	JLE rel8 JNG rel8	ZF = 1|SF<>OF	Короткий, если меньше или равно (не больше)
7F cb	JG rel8 JNLE rel8	ZF = 0&SF = OF	Короткий, если больше (не меньше или равно)
E3 cb	JCXZ rel8 JECXZ rel8		Короткий, если CX/ECX = 0
0F 80 cw/cd	JO rel16/32	OF = 1	Близкий, если переполнение
0F 81 cw/cd	JNO rel 16/32	OF = 0	Близкий, если нет переполнения
0F 82 cw/cd	JB rel16/32 JC rel16/32 JNAE rel 16/32	CF = 1	Близкий, если ниже Близкий, если перенос Близкий, если не выше или равно
0F 83 cw/cd	JAE rel 16/32 JNB rel 16/32 JNC rel 16/32	CF = 0	Близкий, если выше или равно Близкий, если не ниже Близкий, если не перенос

продолжение ⇨

Таблица П.8 *(продолжение)*

Машинный код	Мнемокод	Флаги	Переход
OF84cw/cd	JE rel 16/32 JZ rel 16/32	ZF = 1	Близкий, если равно Близкий, если нуль
OF85cw/cd	JNZ rel16/32 JNErell6/32	ZF = 0	Близкий, если не нуль Близкий, если не равно
OF86cw/cd	JBE rel16/32 JNA rel 16/32	CF = 1|ZF = 1	Близкий, если ниже или равно (невыше)
OF87cw/cd	JA rel16/32 JNBE rell6/32	CF = 0&ZF = 0	Близкий, если выше (не ниже илиравно)
OF88cw/cd	JS rel16/32	SF = 1	Близкий, если знак
OF89cw/cd	JNS rel 16/32	SF = 0	Близкий, если не знак
OF8Acw/cd	JP rel 16/32 JPE rel16/32	PF = 1	Близкий, если паритет (четное количество единичных битов)
OF8Bcw/cd	JPO rell6/32 JNP rel 16/32	PF = 0	Близкий, если не паритет (нечетное количество единичных битов)
0F8C	JL rel 16/32 JNGE rel 16/32	SF<>OF	Близкий, если меньше (не больше или равно)
OF8Dcw/cd	JGE rel 16/32 JNL rel 16/32	SF = OF	Близкий, если больше или равно (не меньше)
OF8Ecw/cd	JLE rell6/32 JNG rel 16/32	ZF = 1|SF<>OF	Близкий, если меньше или равно (не больше)
OF8Fcw/cd	JG rel 16/32 JNLE rel 16/32	ZF - O&SF = OF	Близкий, если больше (не меньше или равно)

JMP

- JMP цель
- EB cb JMP rel8
  E9 cw JMP rell6
  E9 cd JMP rel32
  FF/4 JMP r/m16
  FF/4 JMP r/m32
  EA cd JMP ptr16:16
  EA cp JMP ptrl6:32
  FF/5 JMP m16:16
  FF/5 JMP m16:32

■ Безусловный переход к цели.

Действие: далее перечислены возможные варианты задания операнда цель:

- rel8/16/32 — короткий относительный переход. Значение rel8/16/32 трактуется как знаковое и является смещением перехода относительно следующей за JMP команды в сегменте кода, то есть адрес цели равен (EIP/IP) + (rel8/16/32);
- r16(32)/m16(32) — близкий абсолютный косвенный переход. Цель — регистр r16(32) или ячейка памяти ml6(32), содержащие адрес перехода в текущем сегменте кода;
- ptr16:16(32) — дальний абсолютный переход. Цель — компоненты полного адреса в виде 4- или 6-разрядного указателя, по которому необходимо произвести переход;
- m16:16(32) — дальний абсолютный косвенный переход. Цель — адрес ячейки памяти размером 32(48) битов со структурой m16:16(32), содержащей компоненты адреса перехода.

Выполнение команды при дальнем переходе зависит от режима работы процессора:

- в реальном режиме или режиме виртуального процессора 8086 команда JMP передает управление по адресу, определяемому операндом цель, который может задаваться прямо (ptr16:16(32)) или косвенно (m16:16(32));
- в защищенном режиме выполняются три типа переходов: дальний переход в подчиненный или неподчиненный сегмент кода; дальний переход через шлюз вызова; переключение задачи.

Команду JMP нельзя использовать для передачи управления между уровнями привилегий.
Флаги: изменяются только при переключении задачи.

LAHF

- LAHF
- 9F
- Загрузка регистра AH содержимым младшего байта регистра флаговEFLAGS/FLAGS.

Действие: EFLAGS/FLAGS(SF:ZF:0:AF:0:PF:1:CF) —> AH, здесь в скобках указаны порядок следования флагов, пересылаемых в AH, и значения полей с фиксированным значением.

LAR

- LAR приемник, источник
- OF 02/r . LAR r16,r/m16
  OF 02 /r LAR r32,r/m32
- Загрузка байта прав доступа AR в регистр общего назначения.

Действие: выясняется доступность дескриптора, селектор которого находится в операнде источник. Если дескриптор недоступен на текущем уровне привилегий или по какой-то другой причине, то флаг ZF устанавливается в 0 и работа команды заканчивается. Если дескриптор доступен, то действия команды зависят от установленного размера операнда:

- если операнд 16-разрядный, из дескриптора извлекаются биты 32...47, которые помещаются в 16-разрядный регистр приемник, и выполняется операция приемник = приемник AND 0ff00h;
- если операнд 32-разрядный, из дескриптора извлекаются биты 32...61, которые помещаются в 32-разрядный регистр приемник, и выполняется операция приемник = приемник AND 00f?ff00h (тетрада «?» не определена).

В табл. П.9 определены допустимые для команды LAR типы дескрипторов сегментов. Тип определяет значение младшей тетрады байта AR в дескрипторе при установленном в единицу бите S.

Таблица П.9. Типы дескрипторов сегментов для команды LAR

Тип	Название	Допустимость
0	Резерв	Нет
1	Доступный 16-разрядный сегмент TSS	Да
2	LDT	Да
3	Занятый 16-разрядный сегмент TSS	Да
4	16-разрядный шлюз вызова	Да
5	16/32-разрядный шлюз задачи	Да
6	16-разрядный шлюз прерывания	Нет
7	16-разрядный шлюз ловушки	Нет
8	Резерв	Нет
9	Доступный 32-разрядный сегмент TSS	Да
A	Резерв	Нет
B	Занятый 32-разрядный сегмент TSS	Да
C	32-разрядный шлюз вызова	Да

продолжение

Таблица П.9 *(продолжение)*

Тип	Название	Допустимость
D	Резерв	Нет
E	32-разрядный шлюз прерывания	Нет
F	32-разрядный шлюз ловушки	Нет

Флаги: ZF = r

LDS/LES/LFS/LGS/LSS

- LDS приемник, источник
  LES приемник, источник
  LFS приемник, источник
  LGS приемник, источник
  LSS приемник, источник
- C5 /r LDS r16,m16:16
  C5 /r LDS r32,m16:32
  0F B2 /r LSS r16,m16:16
  0F B2 /r LSS r32,m16:32
  C4/r LES r16,m16:16
  C4/r LES r32,m16:32
  0F B4 /r LFS r16,m16:16
  0F B4 /r LFS r32,m16:32
  0F B5 /r LGS r16,m16:16
  0F B5 /r LGS r32,m16:32
- Загрузка из памяти полного указателя.

Действие: в зависимости от действующего режима адресации (use16 или use32) загружаются первые два (четыре) байта из ячейки памяти источник в 16(32)-разрядный регистр, указанный операндом приемник. Следующие два байта в источнике должны содержать сегментную составляющую некоторого адреса или селектор — они в зависимости от используемой команды загружаются в регистр DS/ES/FS/GS/SS.

LEA

- LEA приемник, источник
- 8D /r LEA r16,m
  8D /r LEA r32,m
- Загрузка эффективного адреса (смещения) операнда источник в приемник.

Действие: как показано в табл. П. 10, действие команды зависит от действующего режима адресации и размера операнда (use16 или use32).

Таблица П. 10. Действие команды LEA

Размер операнда	Размер адреса	Действие
16	16	Вычисляется 16-разрядный эффективный адрес источника и сохраняется в 16-разрядном регистре приемника
16	32	Вычисляется 32-разрядный эффективный адрес источника. Младшие 16 бит этого адреса сохраняются в 16-разрядном регистре приемника
32	16	Вычисляется 16-разрядный эффективный адрес источника. Этот адрес расширяется нулем и сохраняется в 32-разрядном регистре приемника
32	32	Вычисляется 32-разрядный эффективный адрес источника и сохраняется в 32-разрядном регистре приемника

LEAVE

- LEAVE
- C9
- Выход из процедуры высокогоуровня.

Действие: команда копирует регистр EBP/BP в ESP/SP, тем самым восстанавливая в регистре ESP/SP то значение, которое было до вызова процедуры. Далее в регистр EBP/BP значение извлекается из новой вершины стека, тем самым восстанавливается кадр вызывающей программы.

LGDT/LIDT

- LGDT/LIDT источник
- 0F 01 /2 LGDT m16&32
  0F 01 /3 LIDT m16&32
- Загрузка регистра глобальной таблицы дескрипторов/дескрипторов прерываний.

Действие: команда LGDT загружает 16 битов размера и 32 бита значения базового адреса начала таблицы GDT в памяти в системный регистр GDTR. Команда LIDT загружает 16 битов размера и 32 бита значения базового адреса начала таблицы IDT в памяти в системный регистр IDTR.

LLDT

- LLDT источник
- 0F 00 /2
- Загрузка регистра локальной таблицы дескрипторов.

Действие: команда выполняет загрузку 16-разрядного регистра LDTR значением из памяти (размером в слово) или 16-разрядного регистра.

LMSW

- LMSW источник
- 0F 01 /6
- Загрузка слова состояния машины (младших 16 бит регистра CR0) значением из слова памяти или 16-разрядного регистра общего назначения.

LOCK

- LOCK
- F0
- Префикс выдачи сигнала LOCK.

Действие: команда LOCK формирует префикс, инициирующий выдачу процессором сигнала блокировки системной шины LOCK. Используется в многопроцессорных конфигурациях, чтобы добиться монопольного владения системной шиной. Сигнал LOCK может формироваться лишь с определенной номенклатурой команд процессора, работающих в цикле чтение—модификация—запись. К их числу относятся: BTS, BTR, BTC, XCHG, ADD, OR, ADC, SBB, AND, SUB, XOR, NOT, NEG, INC, DEC. Команда BTS всегда формируется с префиксом LOCK.

LODS/LODSB/LODSW/LODSD

- LODS источник
  LODSB/LODSW/LODSD
- AC LODS DS:(E)SI
  AD LODS DS:SI
  AD LODS DS:ESI
  AC LODSB
  AD LODSW
  AD LODSD

- Загрузка строки байтов (слов, двойных слов).

Действие: команда загружает элемент из последовательности (цепочки) в регистр-аккумулятор AL/AX/EAX. Адрес элемента содержится в паре DS:ESI/SI. Кроме операции извлечения элемента команда изменяет значение SI на величину, равную длине элемента цепочки. Знак этой величины зависит от состояния флага DF: DF = 0 — значение положительное, DF = 1 — значение отрицательное.

LOOP/LOOPcc

- LOOP/LOOPcc метка
- E2cb LOOP rel8
  El cb LOOPE\LOOPZ rel8
  EOcb LOOPNE\LOOPNZ rel8
- Управление циклом со счетчиком в регистре CX\ECX.

Действие LOOP: выполнить декремент содержимого регистра CX\ECX; проанализировать регистр CX\ECX: если CX\ECX = 0, передать управление следующей за LOOP команде, если CX\ECX>0, передать управление команде, метка которой указана в качестве операнда. Смещение метки относительно текущего значения регистра IP\EIP должно быть в диапазоне −128...+127 байт.

Действие LOOPcc: выполнить декремент содержимого регистра CX\ECX; проанализировать регистр CX\ECX и флаг ZF. Если CX\ECX = 0, передать управление следующей за LOOPxx команде, если CX\ECX>0, передать управление команде, метка которой указана в качестве операнда LOOPxx. Если ZF = 0, то для команд LOOPE/LOOPZ это означает выход из цикла, а для команд LOOPNE/LOOPNZ — переход к началу цикла. Если ZF = 1, то для команд LOOPE/LOOPZ это означает переход к началу цикла, а для команд LOOPNE/LOOPNZ — выход из цикла.

LSL

- LSL приемник, источник
- OF 03 /r LSL r16,r/m16
  OF 03 /r LSL r32,r/m32
- Загрузка размера сегмента.

Действие: извлечь из дескриптора, селектор которого содержит источник, значение размера сегмента и загрузить его в 16/32-разрядный регистр приемник. При этом проверяются условия: селектор не нулевой; селектор видим на текущем уровне привилегий; значение селектора актуально для текущих пределов дескрипторной таблицы GDT или LDT; тип дескриптора допустим в команде LSL (см. табл. П.9). Если эти условия выполняются, то флаг ZF устанавливается в 1 и в приемник загружается значение размера сегмента (в байтах). Если эти условия не выполняются, то флаг ZF устанавливается в 0 и приемник не изменяется.

Флаги: ZF = r

LTR

- LTR источник
- OF 00 /3
- Загрузка регистра задачи.

Действие: помещение в регистр TR содержимого источника, который представляет собой селектор сегмента TSS. После этого сегмент TSS отмечается занятым, для чего устанавливается бит A в байте AR.

MOV

- MOV приемник, источник
- 88 /r MOV r/m8,r8
  89 /r MOV r/m16,r16
  89 /r MOV r/m32,r32

8A /r MOV r8,r/m8
8B /r MOV r16,r/m16
8B /r MOV r32,r/m32
8C /r MOV r/m16,Sreg**
8E /r MOV Sreg,r/m16
A0 MOV AL,moffs8*
A1 MOV AX,moffs16*
A1 MOV EAX,moffs32*
A2 MOV moffs8*,AL
A3 MOV moffs16*,AX
A3 MOV moffs32*,EAXB0 + rb MOV r8,imm8
B8 + rw MOV r16,imm16
B8 + rd MOV r32,imm32
C6 /0 MOV r/m8,imm8
C7 /0 MOV r/m16,imm16
C7/0 MOV r/m32,imm32

- Копирование содержимого операнда источник в операнд приемник.

- MOV источник

- 0F 22 /r MOVCR0,r32
  0F 22 /r MOV CR2,r32
  0F 22 /r MOV CR3,r32
  0F 22 /r MOV CR4,r32
  0F 20 /r MOV r32,CR0
  0F 20 /r MOV r32,CR2
  0F 20 /r MOV r32,CR3
  0F 20 /r MOV r32,CR4
  0F 21/r MOV r32, DR0-DR7
  0F 23 /r MOV DR0-DR7, r32

- Копирование операнда источник в системный (отладочный) регистр или из системного (отладочного) регистра.

Флаги: OF = ? SF = ? ZF = ? AF = ? PF = ? CF = ?

MOVS/MOVSB/MOVSW/MOVSD

- MOVS приемник, источник
  MOVSB/MOVSW/MOVSD

- A4 MOVS ES:(E)DI, DS:(E)SI
  A5 MOVS ES:DI,DS:SI
  A5 MOVS ES:EDI, DS:ESI
  A4 MOVSB
  A5 MOVSW
  A5 MOVSD

- Пересылка строк байтов (слов, двойных слов).

Действие: команда копирует байт, слово или двойное слово из операнда источник в операнд приемник, при этом адреса элементов предварительно должны быть загружены: адрес источника — в пару регистров DS:ESI/SI (DS по умолчанию, допускается замена сегмента); адрес приемника — в пару регистров ES:EDI/DI (замена сегмента не допускается).

В зависимости от состояния флага DF команда изменяет значение регистров ESI/SI и EDI/DI:

- если DF = 0, то содержимое этих регистров увеличивается на длину структурного элемента последовательности;
- если DF = 1, то содержимое этих регистров уменьшается на длину структурного элемента последовательности.

Для пересылки нескольких следующих друг за другом элементов необходимо использовать префикс REP.

MOVSX

- MOVSX приемник, источник
- 0F BE /r MOVSX r16,r/m8
  0F BE /r MOVSX r32,r/m8
  0F BF /r MOVSX r32,r/m16
- Пересылка со знаковым расширением.

Действие: команда преобразует операнд со злаком в эквивалентный ему операнд со знаком большей размерности. Для этого содержимое операнда источник начиная с младших разрядов записывается в операнд приемник. Старшие биты операнда приемник заполняются значением знакового разряда операнда источник.

MOVZX

- MOVZX приемник, источник
- 0F B6 /r MOVZX r16,r/m8
  0F B6 /r MOVZX r32,r/m8
  0F B7 /r MOVZX r32,r/m16
- Пересылка с пулевым расширением.

Действие: команда преобразует операнд без знака в эквивалентный ему операнд без знака большей размерности. Для этого содержимое операнда источник, начиная с его младших разрядов, записывается в операнд приемник. Старшие разряды операнда приемник заполняются двоичным нулем.

MUL

- MUL множитель
- F6 /4 MUL r/m8
  F7 /4 MUL r/m16
  F7 /4 MUL r/m32
- Целочисленное умножение без учета знака.

Действие: команда выполняет умножение без учета знаков. Явно задается один из множителей. Второй множитель задается неявно в регистре AL\AX\EAX (это местоположение фиксировано). Местоположение результата умножения определяется кодом операции и размером множителей (табл. П.11).

Таблица П. 11. Местоположение множителей и результата при выполнении команды MUL

Размер операндов	Первый множитель	Второй множитель	Результат
Байт	AL	r/m8	AX
Слово	AX	г/т 16	DX:AX
Двойное слово	EAX	r/m32	EDX:EAX

Флаги: если старшая половина результата нулевая: OF = CF = O SF = ? ZF = ? AF = ? PF = ?, если старшая половина результата ненулевая: OF = CF = 1 SF = ? ZF = ? AF = ? PF = ?.

NEG

- NEG приемник
- F6/3 NEG r/m8
  F7/3 NEG r/m16
  F7 /3 NEG r/m32

- Изменение знака.

Действие: команда вычисляет двоичное дополнение операнда приемник.

Флаги:

- если приемник равен нулю: CF= 0 OF=r SF=r ZF=r AF=r PF=r;
- если приемник не равен нулю: CF=1 OF=r SF=r ZF=r AF=r PF=r.

NOP

- NOP
- 90

- Нет операции.

Действие: отсутствует. Единственный эффект от использования команды NOP — инкремент регистра EIP.

NOT

- NOT приемник
- F6 /2 NOT r/m8
  F7 /2 NOT r/m16
  F7/2 NOT r/m32
- Инвертирование всех битов операнда приемник.

OR

- OR приемник, маска
- 0C ib OR AL,imm8
  0D iw OR AX,imm16
  0D id OR EAX,imm32
  80 /1 ib OR r/m8,imm8
  81 /1 iw OR r/m16,imm16
  81 /1 id OR r/m32,imm32
  83 /1 ib OR r/m16,imm8
  83 /1 ib OR r/m32,imm8
  08 /r OR r/m8,r8
  09 /r OR r/m16,r16
  09 /r OR r/m32,r32
  0A /r OR r8,r/m8
  0B /r OR r16,r/m16
  0B /r OR r32,r/m32
- Логическое включающее ИЛИ.

Действие: команда выполняет операцию логического ИЛИ над соответствующими парами битов операндов приемник и маска, то есть приемник = приемник OR маска.

Флаги: CF = OF = 0 SF = r ZF = r AF = ? PF = r

OUT

- OUT номер_порта, аккумулятор
- E6 ib OUT imm8, AL
  E7 ib OUT imm8, AX(EAX)
  EE OUT DX, AL
  EF OUT DX, AX(EAX)
- Вывод значения в порт ввода-вывода.

Действие: вывод значения из регистра AL/AX/EAX (аккумулятор) в порт ввода-вывода, номер которого определяется операндом номер_порта.

OUTS/OUTSB/OUTSW/OUTSD

- OUTS порт, источник
  OUTSB/OUTSW/OUTSD
- 6E OUTS DX, DS:(E)SI
  6F OUTS DX, DS:SI(ESI)
  6F OUTS DX, DS:
  6E OUTSB
  6F OUTSW\OUTSO
- Вывод строки байтов (слов, двойных слов) в порт.

Действие: команда копирует элементы размером байт (слово, двойное слово) из источника в порт ввода-вывода. Номер порта ввода-вывода загружается в регистр DX. Адрес ячейки памяти, из которой копируются данные, содержится по адресу DS:ESI/SI (допускается замена сегмента). По результатам команды значение регистров ESI/SI изменяется на длину элемента. Направление изменения зависит от состояния флага DF:

- если DF = 0, то содержимое регистров ESI/SI увеличивается на длину элемента последовательности;
- если DF = 1, то содержимое регистров ESI/SI уменьшается на длину элемента последовательности.

Для пересылки последовательности элементов необходимо использовать префикс REP.

PAUSE

- PAUSE
- F3 90
- Улучшение выполнения циклов ожидания-занятости.

Действие: улучшить выполнение циклов ожидания-занятости (spin-wait loops). При выполнении подобных циклов процессор Pentium 4 испытывает трудности в завершении цикла, обнаруживая возможное нарушение доступа к памяти. Команда PAUSE подсказывает процессору, что данная кодовая последовательность — цикл ожидания-занятости. Процессор использует эту подсказку, чтобы игнорировать возможную ситуацию нарушения доступа к памяти в большинстве случаев. Это улучшает работу процессора вплоть до того, что значительно уменьшает его энергопотребление. По этой причине рекомендуется включать команду PAUSE во все циклы ожидания-занятости.

POP

- POP приемник
- 8F/0 POP m16
  8F/0 POP m32
  58 + rw POP r16
  58 + rd POP r32
  1F POP DS
  07 POP ES
  17 POP SS
  0F A1 POP FS
  0F A9 POP GS
- Извлечение значения из стека.

Действие: команда восстанавливает содержимое вершины стека в регистр, ячейку памяти или сегментный регистр, после чего содержимое ESP/SP увеличивается на четыре байта для use32 и на два байта для use16. Недопустимо восстановление значения в сегментный регистр CS.

POPA/POPAD

- POPA/POPAD

- Восстановление содержимого регистров общего назначения из стека.

Действие: команда POPA/POPAD восстанавливает содержимое всех регистров общего назначения (DI/EDI, SI/ESI, BP/EBP, SP/ESP, BX/EBX, DX/EDX, CX/ECX, AX/EAX) из стека, после чего значение указателя стека SP/ESP увеличивается на 16(32). Содержимое DI/EDI восстанавливается первым. Содержимое SP/ESP при этом не восстанавливается. Какие именно регистры — 16- или 32-разрядные — извлекаются из стека, зависит от установленного размера операнда. При необходимости изменения размера операнда для инструкции POPA/POPAD можно использовать префикс 66h.

POPF/POPFD

- POPF/POPFD
- 9D
- Извлечение регистра флагов из стека.

Действие: команда POPF/POPFD восстанавливает из стека содержимое регистра флагов FLAGS/EFLAGS, после чего увеличивает значение регистра-указателя стека SP/ESP на 2(4). Размерность выталкиваемого регистра зависит от установленного размера операнда. Действие команды POPF/POPFD зависит от режима работы процессора.

- В защищенном режиме на уровне привилегий 0 (или в реальном режиме) в регистре FLAGS/EFLAGS могут быть изменены любые незарезервированные флаги (за исключением флагов VIP, VIF и VM). Флаги VIP и VIF очищаются, а флаг VM не изменяется.
- В защищенном режиме на уровне привилегий, большем чем 0, но меньшем (или равным) значения в поле IOPL, все флаги могут быть изменены за исключением поля IOPL и флагов VIP, VIF и VM. При этом поле IOPL и флаг VM не изменяются, а флаги VIP и VIF очищаются. Флаг IF изменяется, если выполнение производится на уровне, равном или меньшем значению поля IOPL, в противном случае исключения не происходит, но и привилегированные биты не изменяются.
- В режиме виртуального процессора 8086 для использования команды POPF/POPFD уровень привилегий I/O (IOPL) должен быть равен 3 (при этом флаги VM, RF, IOPL, VIP и VIF не изменяются). Если уровень привилегий меньше 3, то команда POPF/POPFD вызывает исключение общей защиты (#GP).

Флаги: изменяются все флаги, кроме зарезервированных и флага VM.

PREFETCHT0/PREFETCHT1/PREFETCHT2/PREFETCHNTA

- PREFETCHT0 источник
  PREFETCHT1 источник
  PREFETCHT2 источник
  PREFETCHNTA источник
- 0F,18,/1 PREFETCHT0 m8
  0F,18,/2 PREFETCHT1 m8
  0F,18,/3 PREFETCHT2 m8
  0F,18,/0 PREFETCHNTA m8
- Предварительная запись в кэш-память данных из оперативной памяти.

Действие: команда проверяет, есть ли строка байтов в кэш-памяти — если есть, то никаких перемещений не производится. Если данных в кэш-памяти нет, то производится перемещение их в кэш-память того уровня, который определяется конкретной командой:

- PREFETCHT0 — загрузка строки в кэш-память всех уровней;
- PREFETCHT1 — загрузка строки в кэш-память всех уровней, за исключением уровня 0;
- PREFETCHT2 — загрузка строки в кэш-память всех уровней, за исключением уровней 0 и 1;
- PREFETCHNTA — загрузка строки в кэш уровня 1.

Количество записываемых байтов зависит от конкретного процессора, но минимальное количество составляет 32 байта.

PUSH

- PUSH источник
- FF /6 PUSH r/m16
  FF /6 PUSH r/m32
  50 + rw PUSH r16
  50 + rd PUSH r32
  6A PUSH imm8
  68 PUSH imm16/imm32
  OE PUSH CS
  16 PUSH SS
  1E PUSH DS
  06 PUSH ES
  OF AO PUSH FS
  OF A8 PUSH GS
- Размещение операнда в стеке.

Действие: команда уменьшает значение регистра-указателя стека SP/ESP на 2(4) и затем записывает значение источника в вершину стека.

PUSHA/PUSHAD

- PUSHA/PUSHAD
- 60
- Записьвсех регистров общего назначения в стек.

Действие: команда уменьшает значение указателя стека ESP/SP на 16(32) (в зависимости от значения атрибута размера адреса — use16 или use32), после чего размещает в стеке регистры общего назначения в следующей последовательности: AX/EAX, CX/ECX, DX/EDX, BX/EBX, SP/ESP, BP/EBP, SI/ESI, DI/EDI (содержимое DI/EDI будет на вершине стека). В стек помещается содержимое SP/ESP, которое было до выполнения команды.

PUSHF/PUSHFD

- PUSHF/PUSHFD
- 9C
- Размещение регистра флагов встеке.

Действие: команда уменьшает значение указателя стека SP/ESP на 2(4), после чего помещает в вершину стека содержимое регистра FLAGS/EFLAGS. При этом флаги VM и RF (биты 16 и 17) не копируются, вместо этого значения для этих флагов в образе EFLAGS, сохраненном на стеке, устанавливаются равными нулю.

RCL/RCR

- RCL/RCR операнд, количество_сдвигов
- D0 /2 RCL r/m8,l
  D2 /2 RCL r/m8,CL
  C0 /2 ib RCL r/m8,imm8
  D1 /2 RCL r/m16,1
  D3 /2 RCL r/m16,CL
  C1 /2 ib RCL r/m16,imm8
  D1 /2 RCL r/m32,l
  D3 /2 RCL r/m32,CL
  Cl /2 ib RCL r/m32,imm8
  D0 /3 RCR r/m8,l
  D2 /3 RCR r/m8,CL
  C0 /3 ib RCR r/m8,imm8

D1 /3 RCR r/m16,1
D3 /3 RCR r/m16,CL
C1 /3 ib RCR r/m16,imm8
D1 /3 RCR r/m32,1
D3 /3 RCR r/m32,CL
Cl /3 ib RCR r/m32,imm8

- Циклический сдвиг операнда влево (вправо) через флаг переноса CF.

Действие: каждый раз при циклическом сдвиге разрядов операнда влево (вправо) старший (младший) его бит становится значением флага переноса CF. Старое содержимое CF вдвигается в операнд справа (слева) и становится значением его младшего (старшего) бита.

Флаги: CF = r OF = ?r

Флаг OF представляет интерес, если сдвиг осуществляется на один разряд. При сдвиге на несколько разрядов его значение не определено (поэтому обозначен ?r). По значению флага OF можно судить о факте изменения знакового (старшего) разряда операнда:

- OF - 1, если после операции сдвига значения флага CF и старшего бита операнда различны;
- OF = 0, если после операции сдвига влево значения флага CF и старшего бита операнда совпадают.

RDMSR

- RDMSR
- 0F 32
- Чтение из регистра MSR.

Действие: команда проверяет то, что уровень привилегий нулевой и что в регистре ECX находится значение, адресующее один из регистров MSR. Если хотя бы одно из этих условий не выполняется, то выполнение команды заканчивается. Если выполняются оба условия, то значение регистра, адресуемого содержимым регистра ECX, помещается в пару 32-разрядных регистров EDX:EAX.

RDPMC

- RDPMC
- 0F 33
- Чтение счетчиков производительности процессора.

Действие: команда проверяет то, что регистр ECX содержит 0 или 1 и бит CR4.PCE = 1 и что бит CR4.PCE = 0 и CPL = 0. Если одно из этих условий выполнено, то содержимое счетчика, идентифицированного содержимым регистра ECX (0 или 1), пересылается в пару регистров EDX:EAX и выполнение команды заканчивается. Если ни одно из этих условий не выполнено, то выполнение команды заканчивается с возбуждением исключения общей защиты.

RDTSC

- RDTSC
- 0F 31
- Чтение 64-разрядного счетчика меток реального времени (Time Stamp Counter, TSC).

Действие: команда проверяет состояние второго бита регистра CR4.TSD (Time Stamp Disable — отключить счетчик меток реального времени):

- если CR4.TSD - 0, то выполнение команды RDTSC разрешается на любом уровне привилегий;
- если CR4.TSD = 1, то выполнение команды RDTSC разрешается только на нулевом уровне привилегий.

Если выполнение команды разрешено на текущем уровне привилегий, то значение 64-разрядного MSR-счетчика TSC сохраняется в паре 32-разрядных регистров EDX:EAX. Если выполнение команды на текущем уровне привилегий запрещено, работа заканчивается.

REP/REPE/REPZ/REPNE/REPNZ

- REP/REPE/REPZ/REPNE/REPNZ
- REP/REPE/REPZ — f3
  REPNE/REPNZ — f2
- Повторение цепочечной операции.

Действие: команды REP/REPE/REPZ/REPNE/REPNZ вызывают повторение цепочечной команды столько раз, сколько указано в регистре ECX, или до тех пор, пока не выполнится обозначенное флагом ZF условие. Префикс REP может быть добавлен к командам INS, OUTS, MOVS, LODS и STOS. Префиксы REPE, REPNE, REPZ и REPNZ могут быть добавлены к командам CMPS и SCAS. Префиксы REPZ и REPNZ — синонимы префиксов REPE и REPNE. Поведение префикса REP не определено при его использовании с командами, отличными от цепочечных.

В табл. П. 12 приведены условия завершения повторения цепочечных команд при использовании с ними определенных префиксов.

Таблица П. 12. Условия завершения повторения цепочечных команд

Префикс повторения	Условие завершения 1	Условие завершения 2
REP	ECX = 0	Нет
REPE/REPZ	ECX = 0	ZF = 0
REPNE/REPNZ	ECX = 0	ZF = 1

Выяснить, какое именно условие привело к завершению выполнения цепочечной команды, можно, анализируя значение в регистре ECX (командой JECXZ) или проверяя флаг ZF (командой *32,* JNZ или JNE).

Флаги: не изменяются (за исключением случая использования префиксов с командами CMPS и SCAS, которые устанавливают флаги в регистре EFLAGS).

RET

- RET
  RET число
- C3 (ret — близкий возврат в вызывающую процедуру);
  CB (ret — дальний возврат в вызывающую процедуру);
  C2 i16 (ret i16 — близкий возврат с выталкиванием i16 байт из стека);
  CA i16 (ret i16 — дальний возврат с выталкиванием i16 байт из стека)
- Ближний (дальний) возврат из процедуры.

Действие: передача управления по адресу, расположенному на вершине стека. Этот адрес обычно помещается в стек командой CALL, его значение соответствует команде, следующей за командой CALL. Необязательный операнд число определяет количество байтов стека, которые будут вытолкнуты после выталкивания адреса возврата. Команда RET используется для выполнения трех вариантов возврата управления.

- Близкий возврат — возврат управления вызывающей процедуре в пределах текущего сегмента кода (внутрисегментный возврат). При этом из вершины стека выталкивается значение в регистр EIP. Регистр CS не изменяется. Процессор продолжает выполнение кода в том же сегменте, но по другому смещению.
- Дальний возврат — возврат управления вызывающей процедуре, расположенной в отличном от текущего сегменте кода (межсегментный возврат). При этом из вершины стека последовательно выталкиваются значения в регистры EIP и CS. Процессор продолжает выполнение кода в другом сегменте.
- Дальний возврат между уровнями привилегий — дальний возврат управления коду на уровне привилегий, отличном от текущего. Этот вид возврата может быть выполнен только в защищенном режиме. Его механизм подобен механизму дальнего возврата за исключением того,

что процессор следит за уровнями привилегий и правами доступа к сегментам кода и стека, которым возвращается управление для определения возможности подобной передачи. Команда RET очищает регистры DS, ES, FS и GS, если они ссылаются на соответствующие сегменты, недоступные на новом уровне привилегий. Так как при межуровневом возврате управления производится переключение стека, то команда RET также производит загрузку этих регистров из стека. Если в вызываемую процедуру передавались параметры, то при возврате в команде RET необходимо указать параметр число для их удаления.

ROL/ROR

- ROL/ROR операнд, количество_сдвигов
- D0 /0 ROL r/m8,1
  D2 /0 ROL r/m8,CL
  C0 /0 ib ROL r/m8,imm8
  D1 /0 ROL r/m16,1
  D3 /0 ROL r/m16,CL
  C1 /0 ib ROL r/m16,imm8
  D1 /0 ROL r/m32,1
  D3 /0 ROL r/m32,CL
  C1 /0 ib ROL r/m32,imm8
  D0 /1 ROR r/m8,1
  D2 /1 ROR r/m8,CL
  C0 /1 ib ROR r/m8,imm8
  D1 /1 ROR r/m16,1
  D3 /1 ROR r/m16,CL
  C1 /1 ib ROR r/m16,imm8
  D1 /1 ROR r/m32,1
  D3 /1 ROR r/m32,CL
  C1 /1 ib ROR r/m32,imm8
- Циклический сдвиг операнда влево (вправо).

Действие: каждый раз при циклическом сдвиге разрядов операнда влево (вправо) его старший выдвигаемый бит вдвигается в операнд справа (слева) и становится одновременно значением младшего (старшего) бита операнда и флага переноса CF.

Флаги: CF = r OF = ?r SF = r ZF = r PF = r AF = ?

Флаг OF представляет интерес, если сдвиг осуществляется на один разряд. При сдвиге на несколько разрядов его значение не определено (поэтому обозначен ?r). По значению флага OF можно судить о факте изменения знакового (старшего) разряда операнда:

- OF = 1, если после операции сдвига значения флага CF и старшего бита операнда различны;
- OF = 0, если после операции сдвига значения флага CF и старшего бита операнда совпадают.

RSM

- RSM
- 0F AA
- Возврат процессора из S-режима с восстановлением контекста.

Действие: команда проверяет контекст прерванной программы на предмет выполнения следующих условий:

- в регистре CR0 недопустимая комбинация значений битов;
- установлен любой из зарезервированных битов в регистре CR4;
- начало области памяти, с которой работает процессор в S-режиме, не выровнено на границу, кратную 32 Кбайт.

Если хотя бы одно из условий выполняется, процессор переходит в состояние *ожидания*. Если ни одно из условий не выполняется, происходит возврат из S-режима и восстановление контекста

из определенной области памяти. Данная операция напоминает восстановление контекста при переключении из вложенной задачи.

Флаги: изменяются в соответствии с содержимым регистра EFLAGS восстанавливаемого контекста.

SAHF

- SAHF
- 9E
- Загрузка регистра флагов EFLAGS/FLAGS изрегистра AH.

Действие: команда загружает значение из младшего байта регистра AH в регистр EFLAGS/FLAGS. При этом флаги SF, ZF, AF, PF и CF инициализируются битами 7, 6, 4, 2, и 0 регистра AH. Биты 1, 3 и 5 регистров EFLAGS/FLAGS не изменяются, то есть остаются равными значениям 1, 0 и 0 соответственно.

Флаги: SF=rZF=rAF=rPF=rCF=r

SAL/SAR

- SAL/SAR операнд, количество_сдвигов
- DO /4 SAL r/m8,I
  D2/4 SAL r/m8,CL
  C0 /4 ib SAL r/m8,imm8
  D1 /4 SAL r/m16,1
  D3 /4 SAL r/m16,CL
  C1 /4 ib SAL r/m16,imm8
  D1/4 SAL r/m32,I
  D3/4 SAL r/m32,CL
  C1 /4 ib SAL r/m32,imm8
  DO /7 SAR r/m8,I
  D2/7 SAR r/m8,CL
  C0 /7 ib SAR r/m8,imm8
  D1 /7 SAR r/m16,1
  D3 /7 SAR r/m16,CL
  C1 /7 ib SAR r/m16,imm8
  DI /7 SAR r/m32,I
  D3/7 SAR r/m32,CL
  C1 /7 ib SAR r/m32,imm8
- Арифметический сдвиг операнда влево (вправо).

Действие: сдвиг всех битов операнда операнд влево (вправо) на количество разрядов, указанных операндом количество_сдвигов, при этом выдвигаемый слева (справа) бит становится значением флага переноса CF. Одновременно для команды SAL справа в операнд вдвигается нулевой бит. Для команды SAR по мере сдвига вправо освобождающиеся места заполняются значением знакового разряда.

Флаги: CF= r OF = ?r SF= r ZF= r PF= r AF = ?

Флаг OF представляет интерес только для команды SAL в случае, если сдвиг осуществляется на один разряд. При сдвиге на несколько разрядов его значение не определено (поэтому обозначен ?r). По значению флага OF можно судить о факте изменения знакового (старшего) разряда операнда:

- если OF = 1, то текущее значение флага CF и значения выдвигаемого слева бита операнда различны;
- если OF = 0, то текущее значение флага CF и значения выдвигаемого слева бита операнда совпадают.

В отличие от других команд сдвига, команда SAR всегда сбрасывает в ноль флаг OF в операциях сдвига на один разряд.

SBB

- SBB операнд_1, операнд_2
- 1C ib SBB AL,imm8
  1D iw SBB AX,imm16
  1D id SBB EAX,imm32
  80 /3 ib SBB r/m8,imm8
  81 /3 iw SBB r/ml6,imml6
  81 /3 id SBB r/m32,imm32
  83 /3 ib SBB r/m16,imm8
  83 /3 ib SBB r/m32,imm8
  18 /r SBB r/m8,r8
  19 /r SBB r/m16,r16
  19 /r SBB r/m32,r32
  1A /r SBB r8,r/m8
  1B /r SBB r16,r/m16
  1B /r SBB r32,r/m32
- Вычитание сзаемом.

Действие: операнд_1 = операнд_1 - (операнд_2 + CF). Состояние флага CF представляет собой заем от предыдущего вычитания. Команда SBB пе различает знаков операндов. Вместо этого процессор устанавливает флаги OF и CF, чтобы идентифицировать факт заема для знакового и беззнакового операндов. Флаг SF отражает знак результата (состояние его старшего бита).

Флаги: CF = r OF = r SF = r ZF = r PF = r AF = r

SCAS/SCASB/SCASW/SCASD

- SCAS приемник
  scasb/scasw/scasd
- AE SCAS ES:(E)DI/SCASB
  AF SCAS ES:(E)DI/SCASW/SCASD
- Сканирование строки байтов (слов, двойных слов).

Действие: команда сканирования SCAS сравнивает путем вычитания значение в регистре EAX/AX/AL и значение в ячейке памяти, локализуемой парой регистров ES:EDI/DI. По результату вычитания устанавливаются флаги (в том числе ZF). Размер сравниваемых элементов зависит от применяемой команды. Чтобы эту команду можно было использовать для поиска значения в последовательности элементов, имеющих размерность байт, слово или двойное слово, необходимо указать один из префиксов REPE или REPNE. Эти префиксы пе только заставляют циклически выполняться команду поиска пока ECX/CX<>0, но и отслеживают состояние флага ZF (см. описание команд REP/REPE/REPNE). Направление просмотра задается флагом DF:

- DF = 0 — просмотр от начала цепочки к ее концу;
- DF = 1 — просмотр от конца цепочки к ее началу.

Флаги: CF = r OF = r SF = r ZF = r PF = r AF = r

SETcc

- SETcc операнд
- Установка байта по условию.

Действие: команда проверяет условие, заданное модификатором в коде операции cc (фактически, состояние определенных флагов, как показано в табл. П.13), и устанавливает операнд логическим значением 01h или 00h в зависимости от истинности этого условия.

SFENCE

- SFENCE
- 0F AE /7

• Гарантированная запись информации из кэш-памяти всех уровней.

Действие: команда сохраняет данные из кэш-памяти по соответствующим адресам оперативной памяти. Выполняется сериализация (сохранение в долговременную память) всех операций записи в память, инициированных перед выдачей команды SFENCE.

Таблица П.13. Условия установки байта командой SETcc

Машинный код	Мнемокод	Флаги	Условие установки байта
OF 90	SETO r/m8	OF = 1	Переполнение
OF 91	SETNO r/m8	OF = 0	Нет переполнения
OF 92	SETB/SETNAE/SETC r/m8	CF = 1	Ниже (не выше или равно) или перенос
OF 93	SETAE/SETNB/SETNC r/m8	CF = 0	Выше или равно (не ниже) или не перенос
OF 94	SETE/SETZ r/m8	ZF = 1	Равно или нуль
OF 95	SETNE/SETNZ r/m8	ZF = 0	Не равно или не нуль
OF 96	SETNA/SETBE r/m8	CF = 1|ZF = 1	Не выше (ниже или равно)
OF 97	SETA/SETNBE r/m8	CF = 0&ZF = 0	Выше (не ниже или равно)
OF 98	SETS r/m8	SF = 1	Знак
OF 99	SETNS r/m8	SF = 0	Нет знака
OF 9A	SETP/SETPE r/m8	PF = 1	Паритет (четное количество единичных битов)
OF 9B	SETNP/SETPO r/m8	PF = 0	Нет паритета (нечетное количество единичных битов)
OF 9C	SETL/SETNGE r/m8	SF<>OF	Меньше (не больше или равно)
OF 9D	SETGE SETNL r/m8	SF = OF	Больше или равно (не меньше)
OF 9E	SETNG/SETLE r/m8	ZF = 1|SF<>OF	Не больше (меньше или равно)
OF 9F	SETG/SETNLE r/m8	ZF = 0|SF = OF	Больше (не меньше или равно)

SGDT/SIDT

• SGOT/SIDT источник

• OF 01 /0 SGDT m
OF 01 /1 SIDT m

• Сохранение регистра глобальной таблицы дескрипторов (таблицы дескрипторов прерываний).

Действие: команда сохраняет содержимое системного регистра GDTR/IDTR в области памяти размером 48 битов. Структурно эти 48 битов представляют 16 битов размера и 32 бита значения базового адреса начала таблицы GDT/IDT в памяти. Если установлен атрибут размера операнда 32 бита, то 16 битов поля предела помещаются в младшее слово области памяти, а 32 бита базового адреса — в старшее двойное слово 48-разрядной области памяти. Если установлен атрибут размера операнда 16 битов, то 16 битов предела помещаются в младшее слово области, и 24 бита базового адреса — в 3-й, 4-й и 5-й байты (6-й байт равен нулю).

SHL/SHR

• SHL/SHR операнд, количество_сдвигов

• D0 /4 SHL r/m8,1
D2 /4 SHL r/m8,CL
C0 /4 ib SHL r/m8,imm8

D1 /4 SHL r/m16,1
D3 /4 SHL r/m16,CL
C1 /4 ib SHL r/m16,imm8
D1 /4 SHL r/m32,I
D3 /4 SHL r/m32,CL
C1 /4 ib SHL r/m32,imm8
D0 /5 SHR r/m8,1
D2 /5 SHR r/m8,CL
C0 /5 ib SHR r/m8,imm8
D1 /5 SHRr/ml6,I
D3 /5 SHR r/m16,CL
C1 /5 ib SHR r/m16,imm8
DI /5 SHR r/m32,I
D3 /5 SHR r/m32,CL
C1 /5 ib SHR r/m32,imm8

- Логический сдвиг операнда влево (вправо).

Действие: сдвиг всех битов операнда операнд влево (вправо) на количество разрядов, указанное операндом количество_сдвигов, при этом выдвигаемый слева (справа) бит становится значением флага переноса CF. Одновременно слева (справа) в операнд вдвигается нулевой бит.

Флага: CF = r OF = ?r SF = r ZF = r PF - r AF = ?

Для команды SHL флаг OF представляет интерес, если сдвиг осуществляется на один разряд. При сдвиге на несколько разрядов его значение не определено (поэтому обозначен ?г). По его значению можно судить об изменении знакового разряда операнда:

- если OF - 1, то текущее значение флага CF и значение выдвигаемого слева бита операнда различны;
- если OF = 0, то текущее значение флага CF и значение выдвигаемого слева бита операнда совпадают.

Для команды SHR флаг CF содержит значение последнего вдвинутого в операнд бита. Флаг OF устанавливается равным старшему значащему биту исходного операнда.

SHLD

- SHLD приемник, источник, количество_сдвигов

- A4 SHLD r/ml6,rl6,imm8
  OF A5 SHLD r/m16,r16,CL
  OF A4 SHLD r/m32,r32,imm8
  OF A5 SHLD r/m32,r32,CL

- Сдвиг двойного слова влево.

Действие: команда сдвигает операнд приемник влево на число битов, указанных операндом количество_сдвигов. Операнд источник обеспечивает биты, которые вдвигаются в приемник справа (начиная с бита 0 приемника). Операнд количество_ сдвигов — целое число без знака, которое может быть непосредственным 8-разрядным значением или содержимым регистра CL.

Флаги: CF = r OF = ?r SF = r ZF = r PF = r AF = ?

Флаг CF заполняется последним битом, сдвинутым из операнда приемник. Флаг OF устанавливается в 1 для одноразрядного сдвига, если изменился знаковый разряд приемника, иначе он равен нулю. Если операнд количество_сдвигов равен нулю, то флаги не изменяются.

SHRD

- SHRD приемник, источник, количество_сдвигов

- OF AC SHRD r/m16,r16,imm8
  OF AD SHRD r/m16,r16,CL
  OF AC SHRD r/m32,r32,imm8
  OF AD SHRD r/m32,r32,CL

- Сдвиг двойного слова вправо.

Действие: сдвиг операнда приемник вправо на число битов, указанное операндом количество_сдвигов. Операнд источник обеспечивает биты, которые вдвигаются в приемник слева (начиная со старшего бита приемника). Операнд количество_сдвигов — целое число без знака, которое может быть непосредственным 8-разрядным значением или содержимым регистра CL.

Флаги: CF = r OF = ?r SF = r ZF = r PF = r AF = ?

Флаг CF заполняется последним битом, сдвинутым из операнда приемник. Флаг OF устанавливается в 1 для одноразрядного сдвига, если изменился знаковый разряд приемника, иначе он равен нулю.

SLDT

- SLDT приемник
- OF00/0
- Сохранение регистра LOTR локальной таблицы дескрипторов в ячейке памяти или регистре.

SMSW

- SMSW приемник
- OF01/4 SMSWr16(32)/m16
- Сохранение слова состояния машины.

Действие: команда сохраняет значение младших 16 битов регистра CRO в слове памяти или в 16(32)-разрядном регистре общего назначения.

STC

- STC
- F9
- Установка флага переноса CF в единицу.

Флаги: CF = 1

STD

- STD
- FD
- Установка флага направления DF в единицу.

Флаги: DF = 1

STI

- STI
- FB
- Установка флага прерывания IF в единицу.

Флаги: IF = 1

STOS/STOSB/STOSW/STOSD

- STOS приемник
  STOSB/STOSW/STOSD
- AA STOS ES:(E)DI
  AA STOSB
  AB STOS ES:(E)DI
  AB STOSW/STOSD
- Сохранение строки байтов (слов, двойных слов).

Действие: команда записывает элемент из регистра AL/AX/EAX в ячейку памяти, адресуемую нарой ES:DI/EDI. После этого значение регистра DI/EDI изменяется на величину, равную длине элемента цепочки. Знак этого изменения зависит от состояния флага DF:

- если DF = 0, то DI/EDI увеличивается;
- если DF = 1, то DI/EDI уменьшается.

STR

- STR приемник
- 0F 00 /1 STR r/m16
- Сохранение регистра задачи TR.

Действие: сохранение регистра задачи TR в слове памяти или 16-разрядном приемнике (регистре или ячейке памяти).

SUB

- SUB операнд_1, операнд_2
- 2C ib SUB AL,imm8
  2D iw SUB AX,imm16
  2D id SUB EAX,imm32
  80 /5 ib SUB r/m8,imm8
  81 /5 iw SUB r/m16,imm16
  81 /5 id SUB r/m32,imm32
  83 /5 ib SUB r/m16,imm8
  83 /5 ib SUB r/m32,imm8
  28/r SUB r/m8,r8
  29 /r SUB r/m16,r16
  29 /r SUB r/m32,r32
  2A/r SUB r8,r/m8
  2B/r SUB r16,r/m16
  2B/r SUB r32,r/m32
- Вычитание.

Действие: команда выполняет целочисленное вычитание: операнд_1 = операнд_2 - операнд_1. Команда SUB не различает знаков операндов. Вместо этого она соответствующим образом устанавливает флаги.

Флаги: OF=r SF=r ZF=r AF=r PF=r CF=r

SYSENTER

- SYSENTER
- 0F 34
- Быстрый переход к точке входа в коде па уровне 0.

Действие: команда SYSENTER предназначена для максимально эффективного перехода к коду на нулевом кольце защиты (CPL=0). Команда устанавливает регистры CS, EIP, SS, ESP значениями, указанными операционной системой в определенных модельно-зависимых регистрах:

- CS - SYSENTER_CS_MSR;
- EIP - SYSENTER_EIP_MSR;
- SS - 8 + SYSENTER_CS_MSR);
- ESP - SYSENTER_ESP_MSR.

Команда SYSENTER всегда передает управление коду защищенного режима на CPL = 0. Команда может быть вызвана из всех режимов за исключением реального. Команда требует выполнения следующих условий со стороны операционной системы:

- целевой селектор CS должен соответствовать 32-разрядному сегменту кода на нулевом кольце, отображенному на плоское пространство адресов 0...4 Гбайт с доступом по чтению и выполнению;
- целевой селектор SS должен соответствовать 32-разрядному сегменту стека кольца 0, отображенному на плоское адресное пространство 0...4 Гбайт с доступом по чтению, записи, причем величина этого селектора (селектора SS кольца 0) должна быть равна сумме (CS) + 8.

Операционная система посредством указанных ранее модельно-зависимых регистров внутри процессора должна обеспечить необходимые значения для регистров CS, EIP, SS и ESP с тем, чтобы корректно передать управление точке входа кода в кольце защиты 0. Эти регистры доступны по чтению и записи посредством команд RDMSR и WRMSR. Адреса модельно-зависимых регистров представлены в табл. П.14.

Таблица П.14. Адреса модельно-зависимых регистров

Имя	Описание	Адрес
SYSENTER_CS_MSR	Целевой селектор CS на кольце 0	174h
SYSENTER_ESP_MSR	Целевой регистр ESP на кольце 0	175h
SYSENTER_EIP_MSR	Целевой регистр EIP точки входа на кольце 0	176h

SYSEXIT

- SYSEXIT приемник, источник
- 0F 35
- Быстрый возврат из кода на уровне 0.

Действие: команда SYSEXIT предназначена для максимально эффективного перехода к коду на кольце защиты 3 (CPL = 3) из кода на кольце защиты О (CPL = 0). Команда SYSEXIT устанавливает регистры CS, EIP, SS, ESP значениями, указанными операционной системой в определенных моделыю-зависимых регистрах или регистрах общего назначения:

- CS - 16 + SYSENTER_CS_MSR;
- EIP — значение из регистра EDX;
- SS - 24 + SYSENTER_CS_MSR;
- ESP — значение из регистра ECX.

Таблица П. 15. Описание модельно-зависимых регистров

Имя	Описание
Селектор CS	Селектор целевого сегмента кода на кольце 3. Его значение равно сумме 16 + (SYSENTER_CS_MSR)
Селектор SS	Селектор целевого сегмента стека SS на кольце 3. Его значение равно сумме 24 + (SYSENTER_CS_MSR)
Регистр EIP	Адрес возврата в коде на кольце 3. Это целевая точка входа, и она назначается значением, содержащимся в регистре EDX
Регистр ESP	ESP возврата на кольце защиты 3. Назначается значением, содержащимся в регистре ECX

Команда SYSEXIT всегда передает управление коду на кольце 3 (CPL = 3) защищенного режима в рамках плоской модели памяти. Команда может быть вызвана только в защищенном режиме при CPL = 0. Команда требует выполнения следующих условий со стороны операционной системы:

- целевой селектор CS должен соответствовать 32-разрядному неподчиненному сегменту кода на кольце защиты 3, отображенному на адресное пространство 0...4 Гбайт, с атрибутами доступа по выполнению и по чтению;

- целевой селектор SS должен соответствовать 32-разрядному сегменту стека на кольце 3, отображенном на адресное пространство 0...4 Гбайт с атрибутами расширяемости вверх и доступа по чтению и по записи.

Операционная система посредством указанных модельно-зависимых регистров внутри процессора должна обеспечить необходимые значения для регистров CS, EIP, SS и ESP с тем, чтобы корректно передать управление точке входа кода в кольце защиты 3. Эти регистры доступны по чтению и записи посредством команд RDMSR и WRMSR (табл. П.15).

TEST

- TEST приемник, источник
- A8ib TEST AL,imm8
  A9 iw TEST AX,imm16
  A9 id TEST EAX,imm32
  F6 /0 ib TEST r/m8,imm8
  F7/0iw TEST r/m16,imm16
  F7 /0 id TEST r/m32,imm32
  84 /r TEST r/m8,r8
  85 /r TEST r/ml6,rl6
  85 /r TEST r/m32,r32
- Логическое сравнение.

Действие: команда выполняет операцию логического умножения над операндами. Результат операции формируется во временной памяти, сами операнды не изменяются: бит результата равен 1, если соответствующие биты операндов равны 1, в остальных случаях бит результата равен 0. Для анализа результата используется флаг ZF, который равен 1, если результат логического умножения равен нулю.

Флаги: OF = 0 CF = 0 SF = r ZF = r PF = r AF = ?

UD2

- UD2
- 0F 0B
- Генерация исключения недействительного кода операции.

VERR/VERW

- VERR/VERW селектор
- 0F 00 /4 VERRr/ml6
- Определение возможности чтения (записи) из сегмента или в сегмент, заданный операндом селектор.

Действие: команда проверяет выполнение следующих условий:

- определен ли селектор в таблицах GDT или LDT;
- указывает ли дескриптор, адресуемый селектором, на сегмент кода или данных (и ни на какой другой);
- является ли сегмент считываемым (для VERW — записываемым).

Затем проверяется уровень привилегий:

- если сегмент кода является подчиненным, то поле DPL его дескриптора может иметь любое значение;
- если сегмент кода не является подчиненным, поле DPL дескриптора должно быть больше (или равно) полей CPL и RPL селектора.

Если проверка по приведенным выше условиям положительна, то флаг ZF устанавливается в 1, иначе — в 0.

Флаги: ZF = r

WAIT

- WAIT
- 9B
- Приостановка работы процессора до поступления сигнала от сопроцессора об окончании обработки последней команды.

  Машинный код: 9b
  Флаги: C0 = C1 = C2 = C3 = ?

WBINVD

- WBINVD
- 0F09
- Очистка кэш-памяти.

Действие: команда очищает кэш-память первого уровня, записывает содержимое кэш-памяти второго уровня в основную память и очищает содержимое кэш-памяти второго уровня.

WRMSR

- WRMSR
- 0F30
- Запись в MSR-регистр.

Действие: команда проверяет то, что уровень привилегий нулевой и что в регистре ECX содержится значение, идентифицирующее один из MSR-регистров. Если хотя бы одно из этих условий не выполняется, выполнение команды завершается. Если выполняются оба условия, то значение пары 32-разрядных регистров EDX:EAX пересылается в 64-разрядный MSR-регистр, номер которого задан в регистре ECX.

XADD

- XADD приемник, источник
- 0F C0/r XADD r/m8,r8
  0F C1/r XADD r/m16,r16
  0F C1/r XADD r/m32,r32
- Обмен и сложение операндов.

Действие: команда XADD обменивает содержимое операнда приемник с операндом источник, затем формирует сумму их значений в операнде приемник. Эта команда может использоваться с префиксом LOCK.

Флаги: CF = r PF = r AF = r SF = r ZF = r OF = r

XCHG

- XCHG операнд_1, операнд_2
- 90 + rw XCHG AX,r16
  90 + rw XCHG r16,AX
  90 + rd XCHG EAX,r32
  90 + rd XCHG r32,EAX
  86 /r XCHG r/m8,r8
  86 /r XCHG r8,r/m8
  87 /r XCHG r/m16,r16
  87 /r XCHG r16,r/m16
  87 /r XCHG r/m32,r32
  87 /r XCHG r32,r/m32
- Обмен значениями между операндами.

XLAT/XLATB

- XLAT адрес_таблицы_байтов
  XLATB
- D7
- Преобразование байта.

Действие: команда вычисляет адрес в памяти: DS:BX + (AL). Далее команда извлекает байт по этому адресу и помещает его в регистр AL. Несмотря на наличие операнда адрес_таблицы_байтов в команде XLAT, адрес последовательности байтов, из которой будет осуществляться выборка байта для подмены в регистре AL, должен быть предварительно загружен в пару DS:BX(EBX). Команда XLAT допускает замену сегмента.

XOR

- XOR приемник, источник
- 34 ib XOR AL,imm8
  35 iw XOR AX,imm16
  35 id XOR EAX,imm32
  80 /6 ib XOR r/m8,imm8
  81 /6 iw XOR r/m16,imm16
  81 /6 id XOR r/m32,imm32
  83 /6 ib XOR r/m16,imm8
  83 /6 ib XOR r/m32,imm8
  30 /r XOR r/m8,r8
  31 /r XOR r/m16,r16
  31 /r XOR r/m32,r32
  32 /r XOR r8,r/m8
  33 /r XOR r16,r/m16
  33 /r XOR r32,r/m32
- Логическое исключающее ИЛИ.

Действие: выполнение операции логического исключающего ИЛИ над парами битов двух операндов: бит результата равен 1, если значения соответствующих битов операндов различны, в остальных случаях бит результата равен 0. Результат помещается в операнд приемник.

Флаги: OF = r CF = r SF = r ZF = r PF = r AF = ?

Команды сопроцессора

F2XM1

- F2XM1
- D9 F0
- Вычисление 2^x - 1.

Действие: исходный операнд (значение степени x в диапазоне $-1 < x < 1$) хранится в регистре ST0. Команда вычисляет значение 2* - 1, а результат помещает в вершину стека сопроцессора.

Флаги (SWR): C1 устанавливается в 1 при переполнении стека или в 0 при антипереполнении (установлены биты SWR.IE и SWR.SF). При возникновении исключения сопроцессора #P (неточный результат) состояние флага показывает направление выравнивания (1 — в большую сторону); C0, C2, C3 не определены.

FABS

- FABS
- D9E1
- Получение абсолютного значения (модуля) числа в ST0.

Действие: знаковый бит ST0 устанавливается в 0.

Флаги (SWR): C1 устанавливается в 1 при переполнении стека или в 0 при антипереполнении (установлены биты SWR.IE и SWR.SF); C0, C2, C3 не определены.

FADDP/FADD/FIADD

- FAD DP
  FADD/FIADD слагаемое_1
  FADD/FADDP слагаемое_1слагаемое_2
- D8 /0 FADD m32real
  DC /0 FADD m64real
  D8 C0 + i FADD ST(0),ST(i)
  DC C0 + i FADD ST(i),ST(0)
  DE C0 + i FADDP ST(i), ST(0)
  DE C1 FADDP
  DA /0 FIADD m32int
  DE/0 FIADD m16int
- Сложение двух значений.

Действие: команды имеют несколько вариантов расположения операндов/

- Для команды без операндов первое слагаемое располагается в регистре ST(0), второе — в регистре ST(1). Команда выполняет сложение: ST(1) = ST(0) + ST(1). Последнее действие — выталкивание значения из регистра ST(0). Результат сложения — в ST(0).
- Для команды с одним операндом первое слагаемое (операнд слагаемое_1) располагается в ячейке памяти m32real/m64real или m16(32)int, второе слагаемое — в регистре ST(0). Команда выполняет сложение: ST(0) = ST(0) + слагаемое_1.
- Для команды с двумя операндами первое слагаемое (операнд слагаемое_1) сохраняется в регистре ST(0)/ST(i), второе слагаемое (операнд слагаемое_2) — в регистре ST(i)/ST(0). Результат сложения помещается в регистр ST(0) для команды FADD ST(0),ST(i) или регистр ST(i) для команды FADD ST(i),ST(0). Последнее действие команды FADDP — выталкивание значения из регистра ST(0), после чего результат сложения оказывается в регистре ST(i - 1).

Флаги (SWR): см. описание команды F2XM1.

FBLD

- FBLD источник
- DF /4 FBLD m80dec
- Загрузка десятичного числа в стек сопроцессора.

Действие: команда преобразует операнд источник, который содержится в памяти в формате двоично-десятичного числа, в расширенный формат и помещает его регистр ST(0).

Флаги (SWR): см. описание команды FABS.

FBSTP

- FBSTP приемник
- DF/6 FBSTP m80bcd
- Сохранение десятичного значения в памяти с выталкиванием,

Действие: исходный операнд приемник находится в регистре ST(0). Команда преобразует исходный операнд в двоично-десятичный формат и записывает его в область памяти приемника размером 10 байт (18 двоично-десятичных цифр). Последнее действие команды — выталкивание исходного операнда из вершины стека.

Замечание: Если в стеке было не целое число, то оно округляется в соответствии со значением в поле CWR.RC.

Флаги (SWR): см. описание команды F2XM1.

FCHS

- FCHS
- D9 E0
- Инвертирование знака значения в ST(0).

FCLEX/FNCLEX

- FCLEX/FNCLEX
- 9B DB E2 FCLEX
  DB E2 FNCLEX
- Сброс флагов исключений в регистре SWR.

Действие: команда сбрасывает флаги в регистре SWR: все флаги исключений — PE, UE, OE, ZE, DE, IE (биты 0...5); флаг суммарной ошибки работы сопроцессора ES (бит 6); флаг ошибки работы стека сопроцессора SF (бит 7); флаг занятости B (бит 15).

Перед обнулением флагов команда FCLEX выполняет проверку условий ошибки с плавающей точкой. Команда FNCLEX перед обнулением флагов не выполняет проверки условий ошибки с плавающей точкой.

Флаги (SWR): C1, C0, C2, C3 не определены.

FCMOVcc

- FCMOVcc приемник, источник
- Условная пересылка данных между регистрами сопроцессора.

Действие: команда проверяет состояние флагов в регистре EFLAGS (табл. П. 16). Если состояние флагов соответствует условиям выполнения команды, то пересылает значение из источника (ST(i)) в приемник (ST(0)). В противном случае выполнение команды заканчивается без пересылки.

Таблица П. 16. Состояние флагов для пересылки значение из источника в приемник

Машинный код	Мнемокод	Состояние флагов
DA C0 + i	FCMOVB ST(0), ST(i)	CF = 1
DA C8 + i	FCMOVE ST(0), ST(i)	ZF - 1
DA D0 + i	FCMOVBE ST(0), ST(i)	CF - 1 или ZF - 1
DA D8 + i	FCMOVU ST(0), ST(i)	PF = 1
DB C0 + i	FCMOVNB ST(0), ST(i)	CF = 0
DB C8 + i	FCMOVNE ST(0), ST(i)	ZF = 0
DB D0 + i	FCMOVNBE ST(0), ST(i)	CF = 0 и ZF = 0
DB D8 + i	FCMOVNU ST(0), ST(i)	PF = 0

Флаги (SWR): см. описание команды FABS.

FCOM/FCOMP/FCOMPP

- FCOM/FCOMP/FCOMPP
  FCOM/FCOMP операнд
- D8 /2 FCOM m32real
  DC /2 FCOM m64real
  D8 D0+i FCOM ST(i)
  D8 D1 FCOM
  D8 /3 FCOMP m32real
  DC /3 FCOMP m64real

D8D8+i FCOMP ST(i)
D8 D9 FCOMP
DE D9 FCOMPP

- Сравнение вещественных чисел.

Действие: сравнение значения в регистре ST(0) и значения операнда, указанного в команде (mem32/64/ST(i)) или принимаемого по умолчанию (регистр ST(1)). Результат сравнения определяется состоянием битов C3, C2 и C0 регистра SWR (табл. П.17). Последняя операция для FCOMP — выталкивание значения из ST(0). Последняя операция FCOMPP — выталкивание значений из ST(0) и ST(1).

Таблица П. 17. Результат сравнения операндов по состоянию битов C3, C2 и C0 регистра SWR

Результат сравнения	Биты C3, C2 и C0 регистра SWR
Операнды несравнимы	C3 = 1, C2 = 1, C0 = 1
Второй операнд больше первого	C3 = 0, C2 = 0, C0 = 1
Второй операнд меньше первого	C3 = 0, C2 = 0, C0 = 0
Второй операнд равен первому	C3 = 1, C2 = 0, C0 = 0

Флаги (SWR): Бит C1 устанавливается в 1 при переполнении стека или в 0 при антипереполнении (установлены биты SWR.IE и SWR.SF); бит C2 устанавливается в 1, если операнды несравнимы; C0, C3 — результат сравнения.

FCOMI/FCOMIP/FUCOMI/FUCOMIP

- FCOMI/FCOMIP/FUCOMI/FUCOMIP операнд_1, операнд_2
- DB F0 + i FCOMI ST, ST(i)
  DF F0 + i FCOMIP ST, ST(i)
  DB E8 + i FUCOMI ST, ST(i)
  DF E8 + i FUCOMIP ST, ST(i)
- Сравнение вещественных значений и установка EFLAGS.

Таблица П. 18. Установка флагов ZF, PF, CF в регистре EFLAGS по результатам сравнения

Результат сравнения	ZF	PF	CF
ST(0)>ST(i)	0	0	0
ST(0)<ST(i)	0	0	1
ST(0)=ST(i)	1	0	0
Не поддерживаемые форматы	1	1	1

Действие:

1. Команда проверяет, поддерживаются ли форматы чисел в операндах операнд_1 и операнд_2:
   - для команд FCOMI/FCOMIP если один или оба операнда — нечисла (NaN) или числа в не поддерживаемом сопроцессором формате, то возбуждается исключение недействительной операции сопроцессора, в результате чего управление передается соответствующему обработчику. Если бит CWR.IM = 1, то устанавливаются флаги ZF = PF = CF = 1. Если оба операнда — корректные вещественные числа, выполнение команды продолжается;
   - для команд FUCOMI/FUCOMIP если один или оба операнда — нечисла (QNaN, но не SNaN) или числа в не поддерживаемом сопроцессором формате, то устанавливаются флаги ZF = PF = CF = 1. Иначе (один или оба операнда — нечисла SNaN или числа в не поддерживаемом формате) возбуждается исключение недействительной операции и, если CWR.IM - 1, устанавливаются флаги ZF = PF = CF = 1. Если оба операнда — корректные вещественные числа, выполнение команды продолжается.

2. Выполнение вычитания (операнд_1 - операнд_2).
3. По результатам вычитания установка флагов ZF, PF, CF в регистре EFLAGS (табл. П.18). Последняя операция для FCOMIP/FUCOMIP — выталкивание значения из ST(0).

Флаги (EFLAGS): ZF = r PF - r CF = r

Флаги (SWR): см. описание команды FABS.

FCOS

- FCOS
- D9 FF
- Вычисление косинуса.

Действия: значение угла в радианах x $(-2^{63} < x \leq +2^{63})$ хранится в регистре ST(0). Команда преобразует число из ST(0) в значение косинуса и записывает его обратно в регистр ST(0).

Флаги (SWR): Флаг C1 устанавливается в 1 при переполнении стека или в 0 при антипереполнении (биты SWR.IE и SWR.SF в регистре установлены). При возникновении исключения #P состояние флага C1 показывает направление выравнивания (1 — в большую сторону). Если C2 = = 1, то флаг C1 не определен. Флаг C2 устанавливается в 1, если значение исходного операнда выходит за границу $|x| \leq 2^{63}$, иначе — C2 = 0. Флаги CO, C3 не определены.

FDECSTP

- FDECSTP
- D9 F6
- Уменьшение указателя вершины стека на единицу.

Действия: если поле SWR.ST - 0, то выполняется присваивание ST = 7, в противном случае выполняется декремент ST = ST - 1.

Флаги (SWR): C1 устанавливается в 0; CO, C2, C3 не определены.

FDIV/FDIVP/FIDIV

- FDIVP
  FDIV/FIDIVделитель
  FDIV/FDIVP делимое, делитель
- D8 /6 FDIV m32real
  DC /6 FDIV m64real
  D8 FO+i FDIV ST(0), ST(i)
  DC F8 + i FDIV ST(i), ST(0)
  DE F8+i FDIVP ST(i), ST(0)
  DE F9 FDIVP
  DA /6 FIDIV m32int
  DE /6 FIDIV m16int
- Деление двух чисел.

Действия: команды FDIV/FDIVP/FIDIV имеют несколько вариантов расположения операндов.

ii Для команды без операндов делимое находится в регистре ST(1), делитель — в регистре ST(0). Команда выполняет деление: ST(1) = ST(1)/ST(0). Последнее действие — выталкивание значения из ST(0). Окончательный результат — в регистре ST(0).

в Для команды с одним операндом (делитель) делимое находится в регистре ST(0), делитель — в ячейке памяти m32(64)real или ml6(32)int. Команда выполняет деление ST(0) = ST(0)/делитель.

ii Для команды с двумя операндами делимое и делитель хранятся в двух регистрах стека, один из которых — ST(0). Выполняется деление делимое - (делимое/делитель).

Флаги (SWR): см. описание команды F2XM1.

FDIVR/FDIVRP/FIDIVR

- FDIVRP
  FDIVR/FIDIVR делимое
  FDIVR/FDIVRP делитель, делимое
- D8 /7 FDIVR m32real
  DC/7 FDIVR m64real
  D8 F8 + i FDIVR ST(0), ST(i)
  DC FO + i FDIVR ST(i), ST(0)
  DE F0 + i FDIVRP ST(i), ST(0)
  DE F1 FDIVRP
  DA /7 FIDIVR m32int
  DE /7 FIDIVRml6int
- Деление двух чисел в обратном порядке.

  Действия: команды имеют три варианта расположения операндов.

  - Для команды без операндов делимое находится в регистр ST(0), делитель — в регистр ST(1). Команда выполняет деление: ST(1) - ST(0)/ST(1). Последнее действие - выталкивание значения из регистра ST(0). Результат операции — в регистре ST(0).
  - Для команды с одним операндом делимое находится в ячейке памяти m32(64)real или m16(32)int, делитель — в регистре ST(0). Команда выполняет деление: ST(0) = делимое/ST(0).
  - Для команды с двумя операндами делимое и делитель хранятся в двух регистрах стека, один из которых — ST(0). Команда выполняет деление делитель = (делимое/делитель).

  Флаги (SWR): см. описание команды FDIV.

FFREE

- FFREE регистр_сопроцессора
- DD C0 + i FFREE ST(i)
- Освобождение регистра стека сопроцессора.

Действия: команда выбирает в регистре тегов TWR двухразрядное поле tt, соответствующее регистру ST(i), и устанавливает в нем значение 11b.

Флаги (SWR): C1, C0, C2, C3 не определены.

FICOM/FICOMP

- FICOM/FICOMP операнд
- DE /2 FICOM m16int
  DA /2 FICOM m32int
  DE/3 FICOMP m16int
  DA /3 FICOMP m32int
- Сравнение целого и вещественного значений.

Действия: сравнение значения в регистре ST(0) и целочисленного операнда в ячейке памяти m16/32int. Результат сравнения определяется состоянием битов C3, C2 и CO регистра SWR сопроцессора (табл. П. 19). Последнее действие FICOMP — выталкивание значения из вершины стека.

Таблица П. **19.** Результат сравнения операндов по состоянию битов C3, C2 и CO регистра SWR

Результат сравнения	Биты C3, C2 и CO регистра SWR
Операнды несравнимы	C3 = 1, C2 = 1, C0 = 1
Второй операнд больше первого	C3 - 0, C2 - 0, CO - 1
Второй операнд меньше первого	C3 - 0, C2 - 0, CO = 0
Второй операнд равен первому	C3 = 1, C2 = 0, C0 = 0

Флаги (SWR): см. описание команды FCOM.

FILD

- FILD источник
- DF /0 FILD m16int
  DB /0 FILD m32int
  DF /5 FILD m64int
- Целочисленная загрузка.

Действия: преобразование целого значения из операнда источник в вещественное расширенное представление, после чего уменьшение на 1 указателя вершины стека сопроцессора (поле SWR.TOP). Результат преобразования помещается в регистр ST(0).

Флаги (SWR): C1 устанавливается в 1 при переполнении стека; CO, C2, C3 не определены.

FINCSTP

- FINCSTP
- D9 F7
- Увеличение величины указателя вершины стека на единицу.

Действия: если поле SWR.TOP = 7, выполняется присваивание SWR.TOP = 0, в противном случае выполняется инкремент SWR.TOP = SWR.TOP + 1.

Флаги (SWR): C1 устанавливается в 0; CO, C2, C3 не определены.

FINIT/FNINIT

- FINIT/FNINIT
- 9B DB E3 FINIT
  DB E3 FNINIT
- Инициализация сопроцессора.

Действия: команда проверяет наличие установленных битов исключений в регистре SWR. Если какие-то из них установлены, то инициирует вызов процессором обработчиков соответствующих исключений (только FINIT); устанавливает регистры сопроцессора: CWR = 03 7f; SWR = 00 00; TWR - ffff; устанавливает в 0 регистры указателей данных DPR и команд IPR.

Флаги (SWR): C1 = C0 = C2 = C3 = 0

FIST/FISTP

- FIST/FISTP приемник
- DF /2 FIST m16int
  DB/2 FIST m32int
  DF /3 FISTP m16int
  DB/3 FISTP m32int
  DF /7 FISTP m64int

И Целочисленное сохранение.

Действия: исходя из значения в поле RC (управление округлением) регистра CWR, команда округляет число в вершине стека до соответствующего целого значения:

- если RC = OOЬ, то округление до ближайшего целого;
- если RC = 01b, то округление до ближайшего меньшего целого;
- если RC = 10Ь, то округление до ближайшего большего целого;
- если RC = 11b, то дробная часть числа отбрасывается.

Результат преобразования помещается в ячейку памяти, адрес которой указан операндом приемник. Если величина преобразованного значения превышает по модулю максимально представимое число в операнде приемник, то в нем формируется наибольшее отрицательное число — 80 00 или 80 00 00 00. Дополнительное действие для FISTP — выталкивание значения из вершины стека.

Флаги (SWR): см. описание команды F2XM1.

FLD1/FLDL2T/FLDL2E/FLDLG2/FLDLN2/FLDPI/FLDZ

- FLD1/FLDL2T/FLDL2E/FLDLG2/ FLDLN2/FLDPI/ FLDZ
- D9E8 FLD1
  D9 E9 FLDL2T
  D9 EA FLDL2E
  D9 EB FLDPI
  D9 EC FLDLG2
  D9 ED FLDLN2
  D9 EE FLDZ
- Загрузка константы.

Действия:

- FLD1 - загрузка вещественной единицы: SWR.TOP = SWR.TOP - 1; ST(0) - +1.0.
- FLDL2T - загрузка двоичного логарифма десяти: SWR.TOP - SWR.TOP - 1; ST(0) = $\log_2 10$.
- FLDL2E - загрузка двоичного логарифма числа с: SWR.TOP = SWR.TOP - 1; ST(0) = $\log_2 e$.
- FLDLG2 (LoaDing) - загрузка десятичного логарифма числа 2: SWR.TOP = SWR.TOP - 1; ST(0) = $= \log_{10} 2$.
- FLDLN2 (LoaDing) — загрузка натурального логарифма числа 2: SWR.TOP = SWR.TOP – 1; ST(0) = ln(2).
- FLDPI - загрузка числа л: SWR.TOP = SWR.TOP - 1; ST(0) = π.
- FLDZ (LoaDing 0) - загрузка нуля: SWR.TOP = SWR.TOP - 1; ST(0) = +0,0.

Флаги (SWR): C1 устанавливается в 1 при переполнении стека; CO, C2, C3 не определены.

FLD

- FLD источник
- D9 /0 FLD m32real
  DD /0 FLD m64real
  DB /5 FLD m80real
  D9 C0+i FLD ST(i)
- Загрузка в стек вещественного значения.

Действия: операнд источник — значение в вещественном формате в памяти m32/m64/m80 или в регистре ST(i). Команда уменьшает значение поля SWR.TOP на единицу; проверяет тег для регистра стека, физический номер которого находится в поле SWR.TOP. Если содержимое тега не равно 11b (регистр не пустой), то устанавливаются биты SWR.SF и SWR.IE и выполнение команды заканчивается. Если содержимое тега равно 11b, то анализируется тип операнда и выполняются следующие действия:

- если операнд источник является вещественным значением в памяти размером 32 или 64 бита, то оно преобразуется в вещественное число в расширенном формате, после чего записывается в вершину стека;
- если операнд источник является вещественным значением в памяти размером 80 бит или регистром стека, то он копируется в вершину стека.

Флаги (SWR): C1 устанавливается в 1 при переполнении стека или в 0 в противном случае; CO, C2, C3 не определены.

FLDCW

- FLDCW источник
- D9 /5 FLDCW m2byte
- Загрузка регистра управления CWR содержимым ячейки памяти источник.

Флаги (SWR): C1 устанавливается в 1 при переполнении стека или в 0 в противном случае; CO, C2, C3 не определены.

FLDENV

- FLDENV источник
- D9 /4 FLDENV m14/28byte
- Загрузка рабочей среды сопроцессора.

Действия: команда определяет режим работы процессора (R, V или P) и размер операнда для текущего сегмента кода, который задается значением атрибута use (use16 или use32):

- если use16, то размер буфера, из которого будет производиться восстановление среды сопроцессора, равен 14 байт;
- если use32, то размер буфера, из которого будет производиться восстановление среды сопроцессора, равен 28 байт.

Далее из области памяти формата m14/m28, начальный адрес которой указан операндом источник, производится запись информации в регистры CWR, SWR, TR, IPR, DPR сопроцессора. Структура области памяти, содержащей образ рабочей среды сопроцессора, показана на рис. 17.21.

Флаги (SWR): C1 устанавливается в 1 при переполнении стека или в 0 в противном случае; C0, C2, C3 не определены.

FMUL/FMULP/FIMUL

- FMULP
  FMUL/FIMUL множитель_1
  FMUL/FMULP множитель_1, множитель_2
- D8/1 FMUL m32real
  DC /1 FMUL m64real
  D8 C8 + i FMUL ST(0), ST(i)
  DC C8 + i FMUL ST(i), ST(0)
  DE C8 + i FMULP ST(i), ST(0)
  DE C9
  DA/1 FIMUL m32int
  DE /1 FIMUL m16int
- Умножение.

Действия: команда имеет три варианта расположения операндов.

- Для команды без операндов (только FMULP) первый множитель находится в регистре ST(1), второй — в регистре ST(0). Команда выполняет умножение ST(1) = ST(1) • ST(0). Последнее действие — выталкивание значения из регистра ST(0). Результат умножения — в ST(0).
- Для команды с одним операндом первый множитель (операнд) находится в ячейке памяти m32(64) или m16(32)int, второй — в регистре ST(0). Команда выполняет умножение ST(0) = = множитель_1 • ST(0);
- Для команды с двумя операндами первый множитель (операнд множитель_1) хранится в ST(0)/ST(i), второй множитель (операнд множитель_2) — в ST(i)/ST(0). Команда выполняет умножение (множитель_1 • множитель_2). Результат умножения помещается либо в регистр ST(0) для команды FMUL ST(0),ST(i), либо в регистр ST(i) для команды FMUL/FMULP ST(i),ST(0).

Флаги (SWR): см. описание команды F2XM1.

FNOP

- FNOP
- D9 D0
- Нет операции.

Действия: команда не выполняет никаких действий. Эффект ее применения *заключается* в том, что она занимает место (два байта) в потоке команд.

Флаги (SWR): C1 = C0 = C2 = C3 = 0

FPATAN

- FPATAN
- D9 F3
- Вычисление частичного арктангенса.

Действие: исходные операнды: значение x в регистре ST(0); значение y в регистре ST(1). Команда вычисляет значение арктангенса частного x/y; выталкивает значения x и y из стека сопроцессора; записывает результат вычисления арктангенса в вершину стека — регистр ST(0).

Исходные значения x и y должны отвечать условию $0 < y < x < +\infty$. Если это не так, необходимо использовать следующие формулы приведения:

$$\mathrm{arctg}(x) = -\mathrm{arctg}(-x);$$

$$\mathrm{arctg}(x) = \pi/2 - \mathrm{arctg}(1/x).$$

Флаги (SWR): см. описание команды F2XM1.

FPREM

- FPREM
- D9 F8
- Вычисление частичного остатка от деления ST(0) на ST(1).

Действия: делимое хранится в ST(0), делитель — в ST(1). Команда вычитает значения полей порядков в ST(0) и ST(1): D = ST(0) - ST(1).

- Если полученная разность d меньше 64, выполняется деление I = ST(0)/ST(1), результат которого округляется путем усечения *к ближайшему меньшему целому;* в ST(0) записывается новое значение, равное ST(0) = ST(0) - (ST(1) • I); бит SWR.C2 устанавливается в 0 (это означает, что в ST(0) получен истинный остаток, удовлетворяющий требованию: остаток < делителя); биты C0, C1, C3 в SWR устанавливаются равными значениям трех младших битов значения частного $I - I_2I_1I_0$.
- Если полученная разность d больше 64, то бит SWR.C2 устанавливается в 1 (это означает, что в ST(0) получен пока только *частичный* остаток, не удовлетворяющий требованию: остаток < < делителя, и потребуется повторить обращение к команде FPREM для получения истинного остатка, удовлетворяющего указанному требованию); переменной я присваивается значение из диапазона 32...63; выполняется деление I = (ST(0)/ST(1))/$2^{(d-n)}$; выполняется округление I путем усечения к ближайшему меньшему целому; в ST(0) записывается значение, равное ST(0) - - (ST(1) · $I \cdot 2^{(d-n)}$).

Флаги (SWR): C0 устанавливается равным биту 2 частного; C1 устанавливается в 0 при переполнении стека или устанавливается равным биту 0 частного; C2 устанавливается в 0, если получен истинный остаток, или в 1, если остаток все еще частичный; C3 устанавливается равным биту 1 частного.

FPREM1

- FPREM1
- D9 F5
- Вычисление частичного остатка от деления ST(0) на ST(1) в соответствии со стандартом IEEE-754.

Действие: делимое хранится в регистре ST(0), делитель — в регистре ST(1). Команда вычитает значения полей порядков в ST(0) и ST(1): D = ST(0) - ST(1).

- *Если* полученная разность d меньше 64, то выполняется деление I = ST(0)/ST(1), результат которого округляется *к ближайшему целому;* в ST(0) записывается новое значение, равное ST(0) = - ST(0) - (ST(1) • I); бит SWR.C2 устанавливается в 0 (это означает, что в ST(0) получен истинный остаток, удовлетворяющий требованию остаток < делителя); биты C0, C1, C3 в SWR устанавливаются равными значениям трех младших битов частного $I - I_2I_1I_0$.

- Если полученная разность d больше 64, то бит SWR.C2 устанавливается в 1 (это означает, что в ST(0) получен пока только *частичный* остаток, не удовлетворяющий требованию: остаток < < делителя, и потребуется повторить обращение к команде FPREM1 для получения истинного остатка, удовлетворяющего указанному требованию); переменной n присваивается значение из диапазона 32...63; выполняется деление I = (ST(0)/ST(1))/$2^{(d-n)}$; выполняется округление I путем усечения к ближайшему меньшему целому; в ST(0) записывается значение, равное ST(0) - – (ST(1) · $I \cdot 2^{(d-n)}$).

Команда FPREM1 используется для точного деления чисел в соответствии с требованиями стандарта на вычисления с плавающей точкой IEEE-754, согласно которому значение остатка должно быть меньше половины модуля (делителя).

Флаги (SWR): С0 устанавливается равным биту 2 частного; С1 устанавливается в 0 при переполнении стека или устанавливается равным биту 0 частного; С2 устанавливается в 0, если получен конечный остаток, или в 1, если остаток все еще частичный; С3 устанавливается равным биту 1 частного.

FPTAN

- FPTAN
- D9 F2
- Вычисление частичного тангенса угла в радианах из регистра ST(0).

Действия: если значение в ST(0) находится в диапазоне $-2^{63}...2^{63}$, бит С2 устанавливается в О, вычисляется тангенс значения в ST(0), результат записывается в ST(0), а в стек записывается вещественная единица. Окончательное содержимое регистров: ST(0) = 1,0, ST(1) — тангенс угла. Если значение в ST(0) не находится в диапазоне $-2^{63}...2^{63}$, бит С2 устанавливается в 1.

Флаги (SWR): см. описание команды FCOS.

FRNDINT

- FRNDINT
- D9 FC
- Округление значения в регистре ST(0) до целого.

Действия: выполняется округление значения в ST(0) в соответствии со значением поля CWR.RC:

- если CWR.RC = 00b, округление до ближайшего целого;
- если CWR.RC = 01b, округление до ближайшего меньшего целого;
- если CWR.RC = 10b, округление до ближайшего большего целого;
- если CWR.RC = 11b, округление отбрасыванием дробной части числа.

Результат округления записывается в регистр ST(0).

Флаги (SWR): см. описание команды F2XM1.

FRSTOR

- FRSTOR источник
- DD /4
- Восстановление полной среды сопроцессора.

Действия: выясняются режим работы процессора (R, V или P) и размер операнда для текущего сегмента кода. Эти значения определяют размер буфера (94 или 108 байт), из которого будет производиться восстановление полной среды сопроцессора. После этого производится запись информации в следующие регистры сопроцессора из области памяти т94/108, начальный адрес которой указан операндом источник:

- регистр управления CWR;
- регистр состояния SWR;
- регистр тегов TR;

- регистр указателя команд IPR;
- регистр указателя данных DPR;
- регистры стека сопроцессора ST(0)...ST(7).

Структура области памяти, формируемая командой FRSTOR, аналогична таковой для команды FNSAVE (см. далее описание команды FNSAVE).

Флаги (SWR): C1, C0, C2, C3 формируются в соответствии с новым содержимым регистра CWR.

FSAVE/FNSAVE

- FSAVE/FNSAVE приемник
- 9B DD /6 FSAVE m94/108byte
  DD /6 FNSAVE m94/108byte
- Сохранение полной среды сопроцессора.

Действия: проверяется наличие незамаскированных исключений в регистре SWR и (только для команды FNSAVE) ожидается окончание обработки незамаскированных исключений. Далее выясняются режим работы процессора (R, V или P) и размер операнда для текущего сегмента кода. Эти значения определяют размер буфера (94 или 108 байт), в который будет производиться сохранение полной среды сопроцессора. Запись информации в область памяти m94/108, начальный адрес которой указан операндом приемник, производится из следующих регистров сопроцессора:

- регистр управления CWR;
- регистр состояния SWR;
- регистр тегов TR;
- регистр указателя команд IPR;
- регистр указателя данных DPR;
- регистры стека сопроцессора ST(0)...ST(7).

Сопроцессор возвращается в начальное состояние путем установки следующих значений регистров сопроцессора: CWR = 037fh; SWR = 0000; TWR = ffff; регистры указателей данных DPR и IPR команд устанавливаются в 0.

Структура области памяти, формируемая командой FSAVE/FNSAVE, показана на рис. 17.20.

Флаги (SWR): C1, C0, C2, C3 сохраняются и очищаются.

FSCALE

- FSCALE
- D9 FD
- Масштабирование значения в ST(0).

Действия: исходные значения: ST(0) $= x$, ST(1) $= y$. Значение y округлить к ближайшему меньшему целому (обозначим это значение d) и вычислить выражение ST(0) $= x \cdot 2^d$. Команда не очищает регистр ST(1).

Флаги (SWR): см. описание команды F2XM1.

FSIN

- FSIN
- D9 FE
- Вычисление синуса значения угла в радианах из регистра ST(0).

Действия: если значение x в регистре ST(0) находится в диапазоне $-2^{63} \le x < +2^{63}$, то присвоить значения CWR.C2 = 0, ST(0) - $\sin(x)$. В противном случае оставить значение в вершине стека без изменений и установить в 1 бит CWR.C2.

Флаги (SWR): см. описание команды FCOS.

FSINCOS

- FSINCOS
- D9 FB
- Вычисление синуса и косинуса значения угла в радианах из регистра ST(0).

Действия: если значение x находится в диапазоне $-2^{63} \leq x \leq +2^{63}$, то присвоить значения CWR.C2 = 0; ST(0) = sin(x); CWR.TOP = CWR.TOP - 1; ST(0) = cos(x). Если x не находится в указанном диапазоне, оставить значение в вершине стека без изменений и установить в 1 бит CWR.C2. Конечный результат сохраняется в регистрах: ST(0) = cos(x), ST(1) = sin(x).

Флаги (SWR): см. описание команды FCOS.

FSQRT

- FSQRT
- D9 FA
- Вычисление корня квадратного из содержимого регистра ST(0) и запись результата в ST(0).

Флаги (SWR): см. описание команды F2XM1.

FST/FSTP

- FST/FSTP приемник
- D9 /2 FST m32real
  DD /2 FST m64real
  DD D0+i FST ST(i)
  D9/3 FSTP m32real
  DD/3 FSTP m64real
  DB/7 FSTP m80real
  DD D8+i FSTP ST(i)
- Сохранение вещественного значения из регистра ST(0).

Действия: если приемник является ячейкой памяти, то перед сохранением производится округление числа в вершине стека до значения, соответствующего размеру мантиссы приемника. Порядок числа также при необходимости приводится к размеру порядка приемника. Режим округления выбирается, исходя из значения в поле CWR.RC. После этого округленное значение из регистра ST(0) копируется в место (ячейку памяти или регистр), указанное операндом приемник. Последнее действие FSTP — выталкивание значения из вершины стека, то есть увеличение его указателя вершины SWR.TOP = SWR.TOP + 1.

Флаги (SWR): см. описание команды F2XM1.

FSTCW/FNSTCW

- FSTCW/FNSTCW приемник
- 9B D9 /7 FSTCW m2byte
  D9 /7 FNSTCW m2byte
- Сохранение управляющего слова сопроцессора.

Действия: управляющее слово сопроцессора CWR сохраняется в слове памяти, указанном операндом приемник, с проверкой (только для команды FSTCW) наличия незамаскированных исключений в регистре SWR.

Флаги (SWR): C1, C0, C2, C3 не определены.

FSTENV/FNSTENV

- FSTENV/FNSTENV приемник
- 9B D9 /6 FSTENV m14/28byte
  D9 /6 FNSTENV m14/28byte
- Сохранение среды сопроцессора в памяти.

Действия: проверяется (только для команды FSTENV) наличие незамаскированных исключений в регистре SWR, затем выясняются режим работы процессора (R, V или P) и размер операнда для текущего сегмента кода. Эти значения определяют размер буфера (14 или 28 байт), в который будет производиться запись среды сопроцессора. В область памяти m14/28, начальный адрес которой указан операндом **приемник**, производится запись информации из следующих регистров сопроцессора:

- регистр управления CWR;
- регистр состояния SWR;
- регистр тегов TR;
- регистр указателя команд IPR;
- регистр указателя данных DPR.

Маски всех исключений устанавливаются в единицу (маскирование исключений). Команда FNSTENV не проверяет наличие незамаскированных исключений в регистре SWR и не ожидает окончания их обработки.

Структура области памяти, сформированная командой, показана на рис. 17.20.

Флаги (SWR): C1, C0, C2, C3 не определены.

FSTSW/FNSTSW

- FSTSW/FNSTSW приемник
- 9B DD /7 FSTSW m2byte
  9B DF E0 FSTSW AX
  DD /7 FNSTSW m2byte
  DF E0 FNSTSW AX
- Сохранение слова состояния сопроцессора.

Действия: сохранение слова состояния сопроцессора (регистра SWR) в слове памяти или регистре AX с проверкой (только для команды FSTSW) наличия незамаскированных исключений в регистре SWR.

Флаги (SWR): C1, C0, C2, C3 не определены.

FSUB/FSUBP/FISUB

- FSUBP
  FSUB/FISUB вычитаемое
  FSUB/FSUBP уменьшаемое, вычитаемое
- D8 /4 FSUB m32real
  DC /4 FSUB m64real
  D8 E0 + i FSUB ST(0), ST(i)
  DC E8 + i FSUB ST(i), ST(0)
  DE E8 + i FSUBP ST(i), ST(0)
  DE E9 FSUBP
  DA /4 FISUB m32int
  DE /4 FISUB m16int
- Вычитание вещественных значений.

Действия: команды имеют варианты расположения операндов.

- Для команды без операндов (FSUBP) в регистре ST(0) находится вычитаемое, в регистре ST(1) — уменьшаемое. Команда выполняет вычитание ST(1) = ST(1) - ST(0). Последнее действие — выталкивание значения из регистра ST(0). Результат вычитания заносится в регистр ST(0).
- Для команды с одним операндом (вычитаемое) в регистре ST(0) находится уменьшаемое, в области памяти m32(64)real или m16(32)int — вычитаемое. Команда выполняет вычитание ST(0) = - ST(0) - вычитаемое.
- Для команды с двумя операндами (уменьшаемое и вычитаемое) уменьшаемое находится в ST(0)/ST(i), вычитаемое — в ST(i)/ST(0). Команда выполняет вычитание (уменьшаемое - вычитаемое)

и помещает результат в регистр ST(0) для команды fsub ST(0),ST(i) или в ST(i) для команды fsub ST(i),ST(0).

Флаги (SWR): см. описание команды F2XM1.

FSUBR/FSUBRP/FISUBR

- FSUBRP
  FSUBR/FISUBR уменьшаемое
  FSUBR/FSUBRP вычитаемое, уменьшаемое
- D8 /5 FSUBR m32real
  OC /5 FSUBR m64real
  D8 E8 + 1 FSUBR ST(0), ST(i)
  DC E0+i FSUBR ST(i), ST(0)
  DE E0 + i FSUBRP ST(i), ST(0)
  DE EI FSUBRP
  DA /5 FISUBR m32int
  DE /5 FISUBR m16int
- Обратное вычитание вещественных значений.

Действия: команды имеют варианты расположения операндов:

- Для команды без операндов (FSUBRP) вычитаемое находится в регистре ST(1), уменьшаемое — в регистре ST(0). Команда выполняет вычитание ST(1) = ST(0) - ST(1). Последнее действие — выталкивание значения из ST(0), после чего результат вычитания оказывается в регистре ST(0).
- Для команды с одним операндом (уменьшаемое) уменьшаемое находится в области памяти m32(64)real или m16(32)int, вычитаемое — в регистре ST(0). Команда выполняет вычитание ST(0) = уменьшаемое - ST(0).
- Для команды с двумя операндами (вычитаемое и уменьшаемое) уменьшаемое помещается в регистр ST(i)/ST(0), вычитаемое — в регистр ST(0)/ST(i). Команда выполняет вычитание (уменьшаемое - вычитаемое) и помещает результат в регистр ST(0) для команды fsubr ST(0),ST(i) или в ST(i) для команды fsubr ST(i),ST(0).

Флаги (SWR): см. описание команды FSUB.

FTST

- FTST
- D9 E4
- Сравнение значения в регистре ST(0) с нулем.

Действия: значение в вершине стека сравнивается с нулем. По результатам работы устанавливаются биты C3, C2, CO в регистре CWR (табл. П.20).

Флаги (SWR): C1 устанавливается в 1 при переполнении стека, иначе сбрасывается в 0; CO, C2, C3 устанавливаются в соответствии с результатами работы команды.

Таблица П.20. Установка битов C3, C2 и CO в регистре SWR по состоянию регистра ST(0)

Состояние регистра ST(0)	Состояние битов C3, C2, CO в регистре CWR
Не определено	C3 = 1, C2 = 1, C0 = 1
ST(0) = 0	C3 = 1, C2 = 0, C0 = 0
ST(0) < 0	C3 - 0, C2 - 0, CO - 1
ST(0) > 0	C3 = 0, C2 - 0, CO - 0

FUCOM/FUCOMP/FUCOMPP

- FUCOM/FUCOMP/FUCOMPP
  FUCOM источник

- DDEO+i FUCOM ST(i)
  DD E1 FUCOM
  DDE8+i FUCOMP ST(i)
  DD E9 FUCOMP
  DA E9 FUCOMPP

- Неупорядоченное сравнение вещественных значений.

Действия: команда имеет два варианта расположения операндов. В команде FUCOM/FUCOMP/FUCOMPP без операндов сравниваемые значения находятся в регистрах стека ST(0) и ST(1). В команде FUCOM с одним операндом сравниваемые значения находятся в регистрах стека ST(0) и ST(i). Команда выполняет сравнение значений, по результатам которого устанавливаются флаги C3, C2, CO в регистре CWR (табл. П.21).

Таблица П.21. Установка битов C3, C2 и CO в регистре SWR по результатам сравнения регистров ST(0) и ST(1)/ST(i)

Значения в регистрах ST(0) **и ST(1)/ST(i)**	Состояние битов **C3**, C2, CO в регистре CWR
ST(0) и ST(l)/ST(i) несравнимы	C3 = 1, C2 = 1, C0 = 1
ST(0) = ST(l)/ST(i)	C3 = 1, C2 = 0, C0 = 0
ST(0) < ST(1)/ST(i)	C3 - 0, C2 = 0, CO - 1
ST(0) > ST(l)/ST(i)	C3 = 0, C2 = 0, CO - 0

Флаги (SWR): C1 устанавливается в 1 при переполнении стека; CO, C2, C3 устанавливаются в соответствии с результатами работы команды.

FWAIT/WAIT

- FWAIT/WAIT
- 9b
- Останов процессора.

Действия: команда приостанавливает работу основного процессора до поступления сигнала от сопроцессора об окончании обработки последней команды.

Флаги (SWR): CO, Cl, C2, C3 не определены.

FXAM

- FXAM
- D9 E5
- Определение типа операнда в регистре ST(0).

Действия: команда определяет тип операнда в ST(0) и, исходя из него, устанавливает биты C3, C2, C1, CO (табл. П.22). В бит C1 помещается знак операнда в ST(0).

Таблица П.22. Определение типа операнда по состоянию битов C3, C2 и CO в регистре SWR

Состояние битов C3, C2, CO	Тип операнда в регистре CWR
C3 = 0, C2 - 0, CO - 0	Неизвестный тип
C3 - 0, C2 = 0, CO = 1	Нечисло
C3 = 0, C2 = 1, C0 = 0	Корректное вещественное число
C3 = 0, C2 = 1, C0 = 1	Бесконечность
C3 - 1, C2 - 0, CO = 0	Нуль
C3 = 1, C2 = 0, CO = 1	Пусто
C3 = 1, C2 = 1, C0 = 0	Денормализованное число

FXCH

- FXCH

  FXCH источник

- D9 C8 + i FXCH ST(i)

  D9 C9 FXCH

- Обмен содержимым регистров стека.

Действия: команда имеет два варианта расположения операндов. В команде FXCH без операндов исходные значения находятся в регистрах стека ST(0) и ST(1). В команде FXCH с одним операндом исходные значения находятся в регистрах стека ST(0) и ST(i). Команда выполняет обмен значениями между содержимым регистров ST(0) и ST(1)/ST(i).

Флаги (SWR): C1 устанавливается в 1 при переполнении стека, иначе сбрасывается в 0; C0, C2, C3 не определены.

FXTRACT

- FXTRACT
- D9 F4
- Выделение порядка и мантиссы значения в ST(0).

Действия: значение порядка выделяется и записывается в регистр ST(1); значение мантиссы выделяется и помещается на вершину стека сопроцессора — регистр ST(0).

Флаги (SWR): см. описание команды FABS.

FYL2X

- FYL2X
- D9 F1
- Вычисление выражения $y \cdot \log_2(x)$.

Действия: значение y хранится в ST(1) = y, x — в ST(0). Команда помещает результат вычисления выражения $y \cdot \log_2(x)$ в регистр ST(0).

Флаги (SWR): см. описание команды F2XM1.

FYL2XP1

- FYL2XP1
- D9 F9
- Вычисление выражения $y \cdot \log_2(x + 1)$.

Действия: значение y хранится в ST(1), x — в ST(0). Значение в ST(1) должно находиться в диапазоне от $-\infty$ до $+\infty$, а в ST(0) — в диапазоне от $-(1 - \sqrt{2}/2)$ до $(1 - \sqrt{2}/2)$. Если значение в ST(0) лежит вне указанного диапазона, то результат операции не определен. Команда записывает результат вычисления выражения $y \cdot \log_2(x + 1)$ в регистр ST(0).

Флаги (SWR): см. описание команды F2XM1.

Команды блока MMX

EMMS

- EMMS
- 0F 77
- Подготовка сопроцессора к исполнению команд.

Действие: команда переводит все поля регистра тегов TWR сопроцессора в единичное состояние.

MASKMOVQ

- MASKMOVQ источник, маска
- 0F F7 /r MASKMOVQ rmmx1,rmmx2
- Запись байтов в память из MMX-регистра по маске (выборочная запись).

Действие: последовательно проверяются знаковые биты всех байтов маски, и если какой-то из этих битов равен 1, то соответствующий байт источника копируется в область памяти, начальный адрес которой содержится в регистрах DS:DI/EDI. Если знаковый бит равен нулю, то соответствующий байт в приемнике будет нулевым.



Особенность работы команды — в том, что при копировании не используется кэш-память.

MOVD

- MOVD приемник, источник
- 0F 6E /r MOVD rmmx, r/m32
  0F 7E /r MOVD r/m32,rmmx
  66 0F 6E /r MOVD rxmm, r/m32
  66 0F 7E /r MOVD r/m32, rxmm
- Перемещение двойного слова.

Действие: у команды есть несколько вариантов действий:

- если приемник является MMX- или XMM-регистром, в биты 0...31 приемника записывается значение источника, остальные биты приемника обнуляются;
- если приемник является 32-разрядной ячейкой памяти или регистром общего назначения, то в приемник записывается значение битов 0...31 источника (MMX- или XMM-регистра).

MOVNTQ

- MOVNTQ приемник, источник
- 0F,E7,/r MOVNTQ m64, rmmx
- Запись 64 бит в память из MMX-регистра (без использования кэш-памяти).

MOVQ

- MOVQ приемник, источник
- 0F 6F /r MOVQ rmmx,rmmx/m64
  0F 7F/r MOVQ rmmx/m64,rmmx
  F3 0F 7E MOVQ rxmm1, rxmm2/m64
  66 0F D6 MOVQ rxmm2/m64, rxmm1
- Перемещение учетверенного слова.

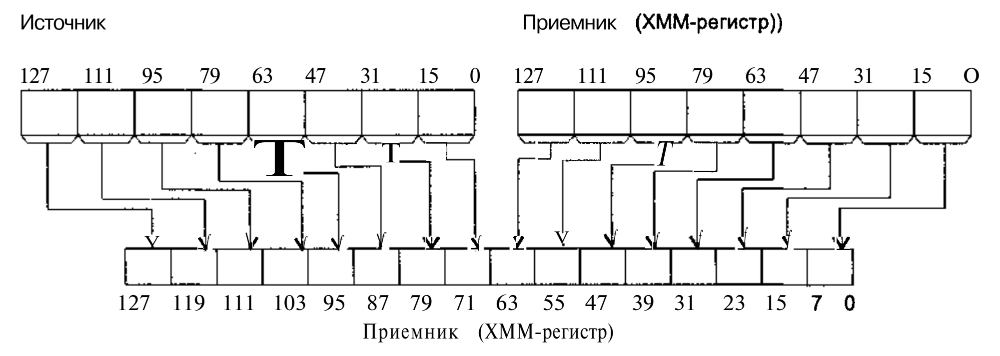
Действие: у команды есть несколько вариантов действий:

- если приемник является MMX-регистром, туда помещается значение источника — MMX-регистра или 64-разрядной ячейки памяти;
- если приемник является 64-разрядной ячейкой памяти, то в разряды 0...63 приемника помещается содержимое разрядов 0...63 источника.
- если приемник и источник являются XMM-регистрами, то в разряды 0...63 приемника помещается содержимое разрядов 0...63 источника, а разряды 64...127 приемника не изменяются.
- если источник является 64-разрядной ячейкой памяти, а приемник — XMM-регистром, то в разряды 0...63 приемника помещается содержимое разрядов 0...63 источника, а в разряды 64...127 приемника - значение 0000000000000000H.

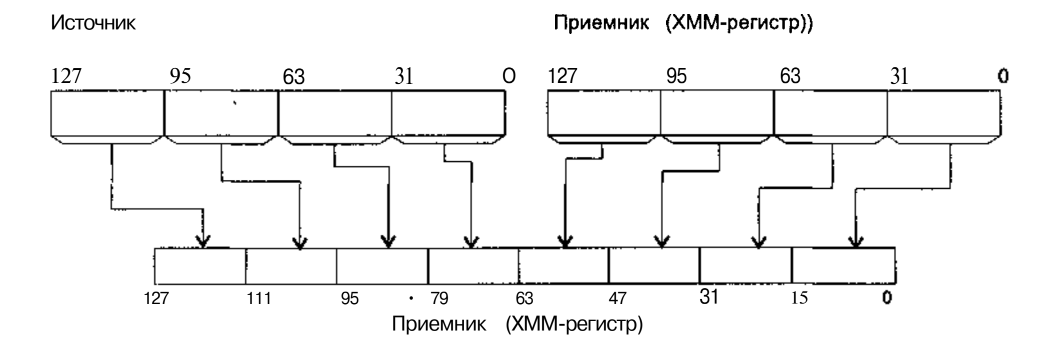
PACKSSWB/PACKSSDW

- PACKSSWB приемник, источник
- 0F 63 /r PACKSSWB mm, mm/m64
  66 0F 63 /r PACKSSWB rxmm1, rxmm2/m128
- Упаковка слов в байты со знаковым насыщением.
- PACKSSDW приемник, источник
- 0F 6B /r PACKSSDW mm, mm/m64
  66 0F 6B /r PACKSSDW rxmm1, rxmm2/m128
- Упаковка двойных слов в слова со знаковым насыщением.

Packsswb *приемник, источник*



Packssdw *приемник, источник*



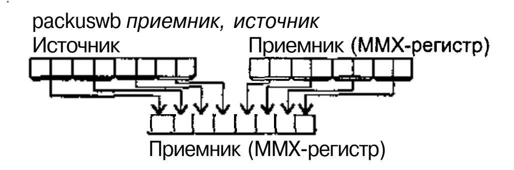
Действие: для MMX-расширения команды преобразуют восемь (четыре) элемента размером в слово (двойное слово) в восемь (четыре) элемента размером в байт (слово). Для XMM-расширения команды преобразуют шестнадцать (восемь) элементов размером в слово (двойное слово) из источника и приемника в шестнадцать/восемь элементов в операнде приемник размером в байт (слово) (см. рис. на предыдущей стр.). Если значение элемента источника превышает допустимое значение элемента приемника, то в элементе приемника формируется предельный результат в соответствии с принципом знакового насыщения:

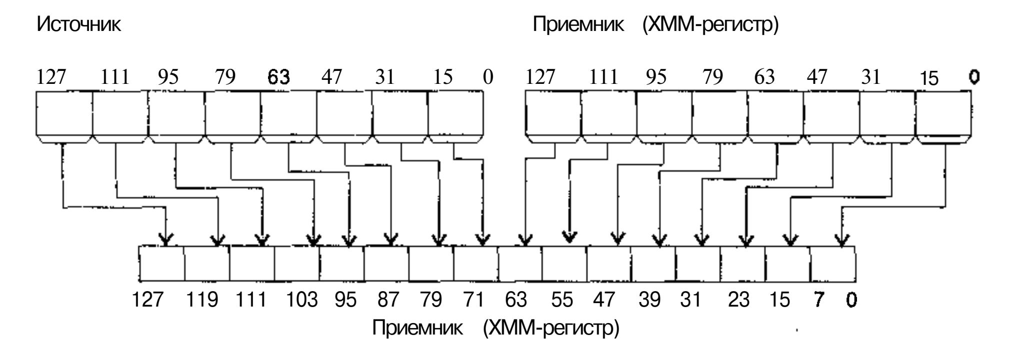
- PACKSSWB — 07fh для положительных чисел и 080h для отрицательных;
- PACKSSDW — 07fffh для положительных чисел и 08000h для отрицательных.

PACKUSWB

- PACKUSWB приемник, источник
- 0f67 /r PACKUSWB rmmx1, rmmx2/m64
  66 0F 67 /r PACKUSWB rxmm1, rxmm2/m128
- Упаковка слов в байты с беззнаковым насыщением.

Действие: команда преобразует восемь (шестнадцать для XMM) элементов размером слово в восемь (шестнадцать для XMM) элементов размером байт (см. рисунок ниже). Если пересылаемое значение больше допустимого для поля приемника, то в нем формируется предельный результат по принципу беззнакового насыщения, что соответствует значениям 0ffh для положительных чисел и 00h для отрицательных.





PADDB/PADDW/PADDD

- PADDB приемник, источник
  PADDW приемник, источник
  PADDD приемник, источник
- 0F FC /r PADDB rmmx1,rmmx2/m64
  66 0F FC /r PADDB rxmm1,rxmm2/m128
  0F FD /r PADDW rmmx1,rmmx2/m64
  66 0F FD /r PADDW rxmm1,rxmm2/m128

OF FE /r PADDD rmmx1,rmmx2/m64
66 OF FE /r PADDD rxmm1,rxmm2/m128

- Сложение упакованных байтов (слов, двойных слов).

Действие: команда в зависимости от кода операции складывает соответствующие элементы операндов источника и приемника размером байт (слово, двойное слово). При возникновении переполнения результат формируется в соответствии с принципом циклического переполнения и помещается в операнд приемник.

PADDQ

- PADDQ приемник, источник
- OF D4 /r PADDQ rmmx1,rmmx2/m64
  66 OF D4 /r PADDQ rxmml,rxmm2/ml28
- Сложение учетверенных слов.

Действие: команда складывает 64-разрядные целые значения в источнике и приемнике. Исходя из типа источника, возможны две схемы умножения:

- если источник — MMX-регистр или 64-разрядная ячейка памяти, то разряды 0...63 приемника складываются с разрядами 0...63 источника и результат помещается в приемник (MMX-регистр);
- если источник — XMM-регистр или 128-разрядная ячейка памяти, то разряды 0...63 приемника складываются с разрядами 0...63 источника и результат помещается в разряды 0...63 приемника, разряды 64...127 приемника складываются с разрядами 64...127 источника и результат помещается в разряды 64...127 приемника.

В результате выполнения команды PADDQ регистр EFLAGS не отражает факта возникновения ситуации переполнения или переноса. Когда результат сложения слишком большой, чтобы быть представленным в 64-разрядном элементе приемника, то он «заворачивается» (перенос игнорируется). Для обнаружения подобных ситуаций программное обеспечение должно использовать другие методы.

Флаги: не изменяются.

PADDSB/PADDSW

- PADDSB приемник, источник
  PADDSW приемник, источник
- OF EC /r PADDSB rmmx1, rmmx2/m64
  66 OF EC /r PADDSB rxmm1, rxmm2/ml28
  OF ED /r PADDSW rmmxl, rmmx2/m64
  66 OF ED /r PADDSW rxmm1, rxmm2/ml28
- Сложение упакованных байтов (слов) со знаковым насыщением.

Действие: команда в зависимости от кода операции складывает соответствующие элементы операндов источник и приемник размером байт (слово) с учетом знака. При переполнении результат формируется в соответствии с принципом знакового насыщения:

- PADDSB — 07fh для положительных чисел и 080h для отрицательных;
- PADDSW — 07fffh для положительных чисел и 08000h для отрицательных.

Результат помещается в операнд приемник.

PADDUSB/PADDUSW

- PADDUSB приемник, источник
  PADDUSW приемник, источник
- OF DC /r PADDUSB rmmxl,rmmx2/m64
  66 OF DC /r PADDUSB rxmm1,rxmm2/m128
  OF DD /r PADDUSW rmmxl,rmmx2/m64
  66 OF DD /r PADDUSW rxmm1,rxmm2/m128

- Сложение упакованных байтов (слов) с беззнаковым насыщением.

Действие: команда в зависимости от кода операции складывает без учета знака соответствующие элементы операндов источник и приемник размером байт/слово. При переполнении результат формируется в приемнике в соответствии с принципом беззнакового насыщения:

- PADDUSB — 0ffh и 00h для результатов сложения, соответственно, больших или меньших максимально/минимально представимых значений в беззнаковом байте;
- PADDUSW — 0ffffh и 0000h для результатов сложения, соответственно, больших или меньших максимально/минимально представимых значений в беззнаковом слове.

Результат помещается в операнд приемник.

PAND

- PAND приемник, источник
- 0F DB /r PAND rmmx1, rmmx2/m64
  66 0F DB /r PAND rxmm1, rxmm2/m128
- Упакованное логическое И.

Действие: команда выполняет поразрядную операцию логического И над всеми битами операндов источник и приемник. Результат помещается в операнд приемник.

PANDN

- PANDN приемник, источник
- 0F DF /r PANDN rmmx1, rmmx2/m64
  66 0F DF /r PANDN rxmm1, rxmm2/m128
- Упакованное логическое И-НЕ.

Действие: команда выполняет поразрядную операцию логического И-НЕ над всеми битами операндов источник и приемник. Результат помещается в операнд приемник.

PAVGB/PAVGW

- PAVGB приемник, источник
  PAVGW приемник, источник
- 0F,E0, /r PAVGB rmmx1, rmmx2/m64
  0F,E3, /r PAVGW rmmx1, rmmx2/m64
  66 0F E0, /r PAVGB rxmm1, rxmm2/m128
  66 0F E3 /r PAVGW rxmm1, rxmm2/m128
- Упакованное среднее.

Действие: команда выполняет параллельное сложение байтов (слов) источника и приемника и сдвигает результат сложения на один разряд вправо (деление на 2).

PCMPEQB/PCMPEQW/PCMPEQD

- PCMPEQB приемник, источник
  PCMPEQW приемник, источник
  PCMPEQD приемник, источник
- 0F 74 /r PCMPEQB rmmx1,rmmx2/m64
  0F 75 /r PCMPEQW rmmx1,rmmx2/m64
  0F 76 /r PCMPEQD rmmx1,rmmx2/m64
  66 0F 74 /r PCMPEQB rxmm1,rxmm2/m128
  66 0F 75 /r PCMPEQW rxmm1,rxmm2/m128
  66 0F 76 /r PCMPEQD rxmm1,rxmm2/m128
- Сравнение на равенство упакованных байтов (слов, двойных слов).

Действие: команды сравнивают на равенство элементы источника и приемника и формируют элементы результата по следующему принципу:

- если элемент источника равен соответствующему элементу приемника, то элемент результата устанавливается в зависимости от применяемой команды равным значению 0ffh, 0ffffh или 0ffffffffh;
- если элемент источника не равен соответствующему элементу приемника, то элемент результата устанавливается в зависимости от применяемой команды равным значению 00h, 0000h или 00000000h.

Результат помещается в операнд приемник.

PCMPGTB/PCMPGTW/PCMPGTD

- PCMPGTB приемник, источник
  PCMPGTW приемник, источник
  PCMPGTD приемник, источник
- 0F 64 /r PCMPGTB rmmx1,rmmx2/m64
  0F 65 /r PCMPGTW rmmx1,rmmx2/m64
  0F 66 /r PCMPGTD rmmx1,rmmx2/m64
  66 0F 64 /r PCMPGTB rxmm1,rxmm2/m128
  66 0F 65 /r PCMPGTW rxmm1,rxmm2/m128
  66 0F 66 /r PCMPGTD rxmm1,rxmm2/m128
- Сравнение по условию «больше чем» упакованных байтов (слов, двойных слов).

Действие: команда производит сравнение по условию «больше чем» элементов операндов источник и приемник и формирует элементы результата по следующему принципу:

- если элемент приемника больше соответствующего элемента источника, то элемент результата устанавливается в зависимости от применяемой команды равным значению 0ffh, 0ffffh или 0ffffffffh;
- если элемент приемника не больше соответствующего элемента источника, то элемент результата устанавливается, в зависимости от применяемой команды, равным значению 00h, 0000h или 00000000h.

Результат помещается в операнд приемник.

PEXTRW

- PEXTRW приемник, источник, маска
- 0F C5 /r ib PEXTRW r32, rmmx, i8
  66 0F C5 /r i8 PEXTRW r32, rxmm, i8
- Извлечение 16-разрядного слова из MMX-регистра по маске.

Действие: команда выделяет два младших бита непосредственного операнда маска. Их значение определяет номер слова в операнде источник (MMX-регистр). Данное слово перемещается в младшие 16 битов операнда приемник, представляющего собой 32-разрядный регистр общего назначения. Старшие 16 бит этого регистра обнуляются.

PINSRW

- PINSRW приемник, источник, маска
- 0F C4 /r ib PINSRW rmmx,r32/m16,i8
  66 0F C4 /r i8 PINSRW rxmm,r32/m16,i8
- Вставка 16-разрядного слова в MMX- или XMM-регистр.

Действие: команда выделяет два младших бита непосредственного операнда маска. Их значение определяет номер слова в операнде приемник, который представляет собой MMX- или XMM-регистр. В это слово будут перемещены младшие 16 битов операнда источник, который представляет собой 32-разрядный регистр общего назначения или 16-разрядную ячейку памяти.

PMADDWD

- PMADDWD приемник, источник

- 0F F5 /r PMADDWD rmmx1,rmmx2/m64
  66 0F F5 /r PMADDWD rxmm1,rxmm2/m128
- Упакованное знаковое умножение знаковых словоперандов источник и приемник с последующим сложением промежуточных результатов в формате двойного слова.

Действие: команда производит умножение с учетом знака четырех знаковых слов источника на четыре знаковых слова приемника и формирует элементы результата.

При переполнении результат устанавливается равным 8000 0000 8000 0000h. Ситуация переполнения возникает, если все элементы обоих операндов равны 8000h.

PMAXSW

- PMAXSW приемник, источник
- 0F EE /r PMAXSW rmmx1,rmmx2/m64
  66 0F EE /r PMAXSW rxmm1,rxmm2/m128
- Возврат максимальных упакованных знаковых слов.

Действие: команда определяет наибольшее слово для каждой пары упакованных слов источника и приемника с учетом знака и заменяет им соответствующие слова приемника.

PMAXUB

- PMAXUB приемник, источник
- 0F DE /r PMAXUB rmmx1,rmmx2/m64
  66 0F DE /r PMAXUB rxmm1,rxmm2/m128
- Возврат максимальных упакованных беззнаковых байтов.

Действие: для каждой пары байтовых элементов источника и приемника определяется наибольший без учета знака и им заменяется соответствующий элемент приемника.

PMINSW

- PMINSW приемник, источник
- 0F EA /r PMINSW rmmx1,rmmx2/m64
  66 0F EA /r PMINSW rxmm1,rxmm2/m128
- Возврат минимальных упакованных знаковых слов.

Действие: для каждой пары элементов (размером 16 бит) источника и приемника команда определяет наименьший с учетом знака и заменяет им соответствующий элемент приемника.

PMINUB

- PMINUB приемник, источник
- 0F DA /r PMINUB rmmx1,rmmx2/m64
  66 0F DA /r PMINUB rxmm1,rxmm2/m128
- Возврат минимальных упакованных беззнаковых байтов.

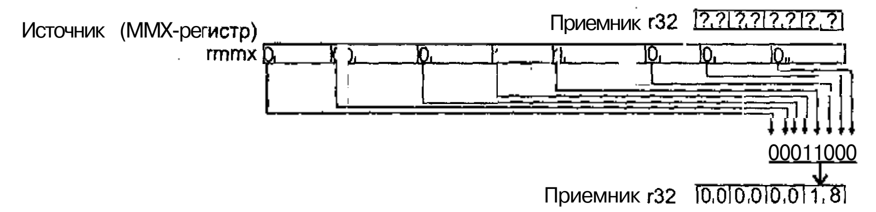
Действие: для каждой пары байтовых элементов источника и приемника команда определяет наименьший без учета знака и заменяет им соответствующий элемент приемника.

PMOVMSKB

- PMOVMSKB приемник, источник
- 0F D7 /r PMOVMSKB r32,rmmx
  66 0F D7 /r PMOVMSKB r32, rxmm
- Перемещение байтовой маски в целочисленный регистр.

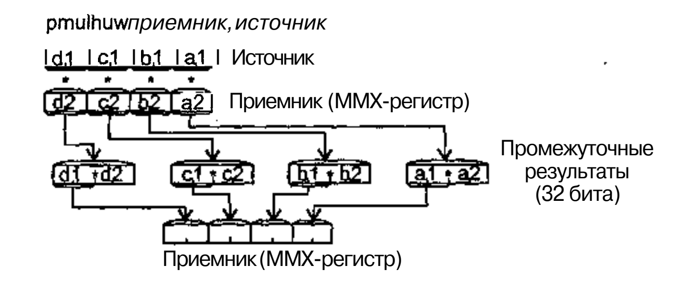
Действие: команда извлекает и копирует значения старшего бита каждого из упакованных байтов MMX- или XMM-регистра в младший байт (слово) 32-разрядного целочисленного регистра общего назначения. Остальные разряды целочисленного регистра обнуляются.

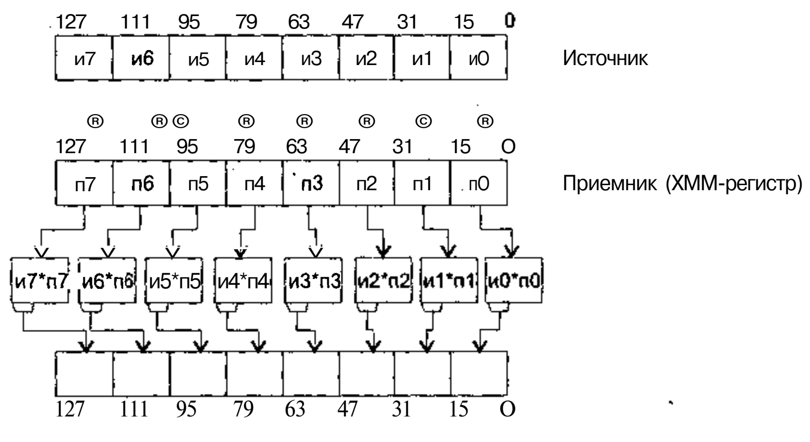
pmovmskb приемник, источник



PMULHUW

- PMULHUW приемник, источник
- 0F E4 /r PMULHUW rmmx1,rmmx2/m64
  66 0F E4 /r PMULHUW rxmm1,rxmm2/m128





Приемник (XMM-регистр)

- Умножение упакованных беззнаковых слов с возвратом старших слов результата.

Действие: команда производит умножение упакованных слов источника и приемника без учета знака и формирует элементы результата в соответствии с приведенной на предыдущей странице схемой. Как видно из нее, в результате умножения слов операндов источник и приемник получаются промежуточные результаты размером 32 бита. Далее старшее слово (16 бит) из каждого промежуточного результата умножения исходных элементов помещается в 16-битный элемент окончательного результата. Результат помещается в операнд приемник.

PMULHW

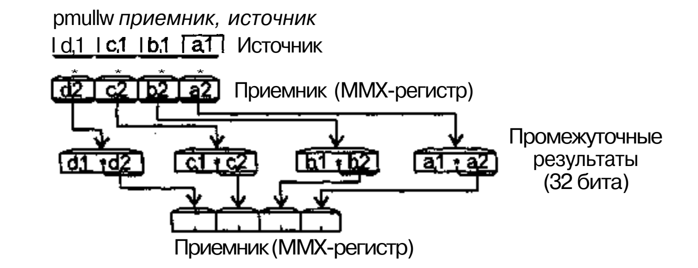
- PMULHW приемник, источник
- OF E5/r PMULHW rmmx1,rmmx2/m64
  66 OF E5 /r PMULHW rxmm1,rxmm2/m128
- Упакованное знаковое умножение слов с возвратом старшего слова результата.

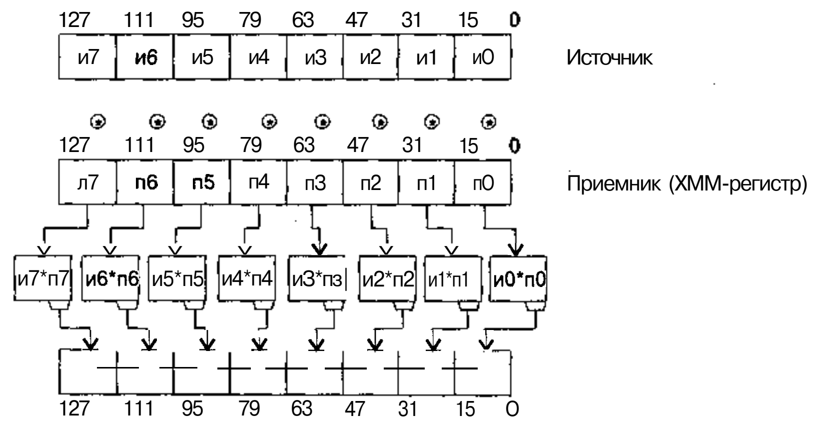
Действие: команда производит умножение упакованных слов источника и приемника с учетом знака и формирует элементы результата в соответствии со схемой, приведенной при описании команды PMULHUW.

PMULLW

- PMULLW приемник, источник
- OF D5 /r PMULLW rmmx1,rmmx2/m64
  66 OF D5 /r PMULLW rxmm1,rxmm2/m128
- Упакованное знаковое умножение слов с возвратом младшего слова результата.

Действие: команда производит умножение с учетом знака упакованных слов источника и приемника и формирует элементы результата в соответствии с приведенной далее схемой.





Как видно из этой схемы, в результате умножения слов источника и приемника получаются промежуточные результаты размером 32 бита. Далее младшее слово (16 битов) из каждого 32-разрядного элемента промежуточного результата умножения исходных элементов помещается в 16-разрядный элемент результата (операнд приемник).

PMULUDQ

- PMULUDQ приемник, источник
- 0F F4 /r PMULUDQ rmmx1,rmmx2/m64
  66 0F F4 /r PMULUDQ rxmm1,rxmm2/m128
- Умножение 32-разрядных целых значений без учета знака и сохранение результата в MMX- или XMM-регистре.

Действие: 32-разрядные целые значения со знаком в источнике и приемнике умножаются. Исходя из типа источника, возможны две схемы умножения:

- если источник — MMX-регистр или 64-разрядная ячейка памяти, то разряды 0...31 источника и приемника перемножаются и помещаются в разряды 0...63 приемника;
- если источник — XMM-регистр или 128-разрядная ячейка памяти, то разряды 0...31 источника и приемника перемножаются и помещаются в разряды 0...63 приемника, а разряды 64...95 источника и приемника перемножаются и помещаются в разряды 64...127 приемника.

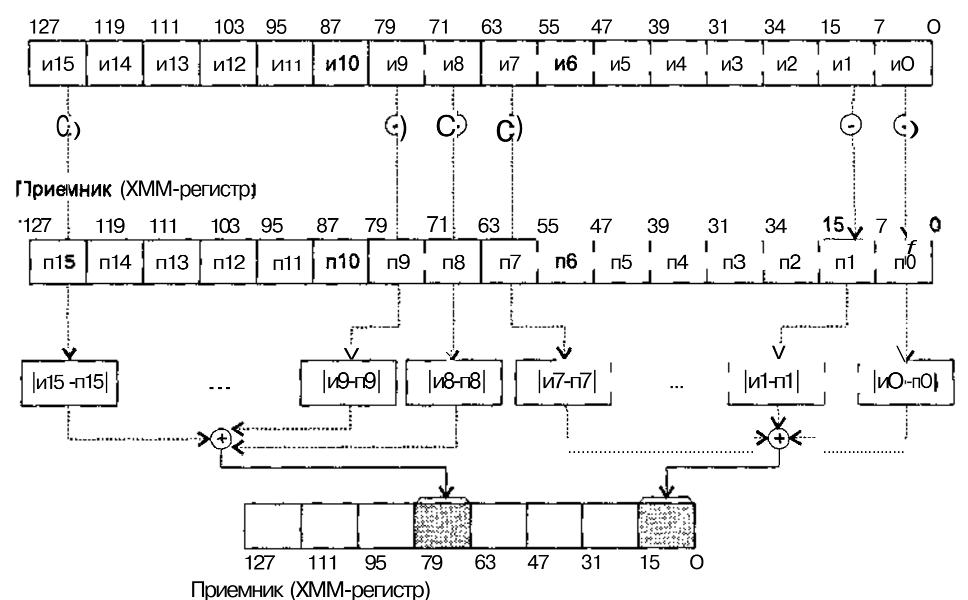
Когда результат умножения слишком большой, чтобы быть представленным в приемнике, то он «заворачивается» (перенос игнорируется).

Флаги: не изменяются.

POR

- POR приемник, источник
- 0F EB /r POR rmmx1,rmmx2/m64
  66 0F EB /r POR rxmm1,rxmm2/m128
- Упакованное логическое ИЛИ.

Действие: команда производит побитовую операцию логического ИЛИ над всеми битами операндов источник и приемник. Результат помещается в операнд приемник.



PSADBW

- PSADBW приемник, источник

■ OF F6 /r PSADBW rmmx1,rmmx2/m64
66 OF F6 /r PSADBW rxmml,rxmm2/ml28

- Суммарная разница значений пар беззнаковых упакованных байтов.

Действие: для каждой пары байтов упакованных учетверенных слов операндов источник и приемник вычисляется модуль разности, после чего полученные модули складываются. Результат записывается в младшее слово упакованного учетверенного слова приемника, старшие три слова приемника обнуляются. Принцип работы команды (XMM) поясняет схема на предыдущей странице.

PSHUFW

- PSHUFW приемник, источник, маска
- 0F 70 /rib PSHUFW rmmx1,rmmx2/m64,i8
- Перераспределение упакованных слов.

Действие: каждая пара битов маски определяет номер слова источника, которое будет перемещено в приемник следующим образом:

▒ Маска[1:0]:

D 00 приемник[00...16] ← источник[00...15];
D 01 приемник[00...16] ← источник[16...31];
D 10 приемник[00...16] ← источник[32...47];
□ 11 приемник[00...16] ← источник[48...63].

it Маска[3:2]:

П 00 приемник[16...31] ← источник[00...15];
П 01 приемник[16...31] ← источник[16...31];
D 10 приемник[16...31] ← источник[32...47];
O 11 приемник[16...31] ← источник[48...63].

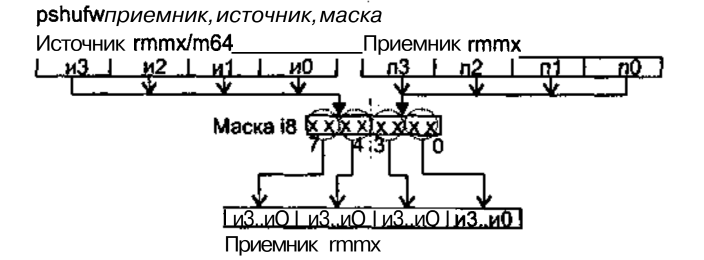
▒ Маска[5:4]:

П 00 приемник[32...47] ← источник[00...15];
P 01 приемник[32...47] ← источник[16...31];
□ 10 приемник[32...47] ← источник[32...47];
O И приемник[32...47] ← источник[48...63].

▒ Маска[7:6]:

П 00 приемник[47...63] ← источник[00...15];
D 01 приемник[47...63] ← источник[16...31];
D 10 приемник[47...63] ← источник[32...47];
O 11 приемник[47...63] ← источник[48...63].

Работа команды PSHUFW показана на следующей схеме.



PSLLW/PSLLD/PSLLQ

- PSLLW приемник, источник
  PSLLD приемник, источник
  PSLLQ приемник, источник
- 0F F1 /r — PSLLW rmmx1, rmmx2/m64
  66 0F F1 /r — PSLLW rxmm1,rxmm2/m128
  0F 71 /6, i8 — PSLLW rmmx1,i8
  66 0F 71 /6 i8 — PSLLW rxmm1,i8
  0F F2 /r — PSLLD rmmx1, rmmx2/m64
  66 0F F2 /r — PSLLD rxmm1, rxmm2/m128
  0F 72 /6 i8 — PSLLD rmmx1,i8
  66 0F 72 /6 ib — PSLLD rxmm1,i8
  0F F3 /r — PSLLQ rmmx1, rmmx2/m64
  66 0F F3 /r — PSLLQ rxmm1, rxmm2/m128
  0F 73 /6 i8 — PSLLQ rmmx1,i8
  66 0F 73 /6 i8 — PSLLQ rxmm1,i8
- Логический сдвиг влево упакованных слов (двойных слов, учетверенных слов).

Действие: команда сдвигает влево биты в элементах размером слово (двойное слово, учетверенное слово) приемника на количество разрядов, определяемое операндом источник. Одновременно справа в каждый элемент вдвигаются нулевые значения. Результат помещается в операнд приемник. Если значение, указанное в операнде источник, больше чем 15 (для слов), 31 (для двойных слов) или 63 (для учетверенного слова), то значение операнда приемник устанавливается равным 0.

PSRAW/PSRAD

- PSRAW приемник, источник
  PSRAD приемник, источник
- 0F E1 /r — PSRAW rmmx1, rmmx2/m64
  66 0F E1 /r — PSRAW rxmm1,rxmm2/m128
  0F 71 /4 ib — PSRAW rmmx1, i8
  66 0F 71 /4 i8 — PSRAW rxmm1,i8
  0F E2 /r — PSRAD rmmx1, rmmx2/m64
  66 0F E2 /r — PSRAD rxmm1,rxmm2/m128
  0F 72 /4 ib — PSRAD rmmx1, i8
  66 0F 72 /4 i8 — PSRAD rxmm1,i8
- Арифметический сдвиг вправо упакованных слов (двойных слов).

Действие: команда сдвигает вправо биты в элементах приемника размером слово (двойное слово) на количество разрядов, определяемое операндом источник. Одновременно слева в разряды каждого элемента вдвигаются биты, количество которых определяется операндом источник. Значение вдвигаемых слева битов равно знаковому биту соответствующего элемента. Результат помещается в операнд приемник. Если значение, указанное в источнике больше чем 15 (для слов), 31 (для двойных слов), то каждый элемент данных приемника заполняется начальным значением знакового разряда элемента.

PSRLW/PSRLD/PSRLQ

- PSRLW приемник, источник
  PSRLD приемник, источник
  PSRLQ приемник, источник
- 0F D1 /r — PSRLW rmmx1,rmmx2/m64
  66 0F D1 /r — PSRLW rxmm1,rxmm2/m128
  0F 71 /2 i8 — PSRLW rmmx1,i8
  66 0F 71 /2 i8 — PSRLW rxmm1,i8

OF D2 /r PSRLD rmmx1,rmmx2/m64
66 OF D2 /r PSRLD rxmm1,rxmm2/m128
OF 72 /2 i8 PSRLD rmmx1,i8
66 OF 72 /2 i8 PSRLD rxmm1,imm8
OF D3 /r PSRLQ rmmx1, rmmx2/m64
66 OF D3 /r PSRLQ rxmm1, rxmm2/m128
OF 73 /2 i8 PSRLQ rmmx1,i8
66 OF 73 /2 i8 PSRLQ rxmm1,i8

- Логический сдвиг вправо упакованных слов (двойных слов, учетверенных слов).

Действие: команда производит сдвиг вправо битов элементов размером слово (двойное слово, учетверенное слово) приемника на количество разрядов, определяемое операндом источник. Одновременно слева в каждый элемент вдвигаются нулевые биты. Результат помещается в операнд приемник. Если значение, указанное в операнде источник, больше чем 15 (для слов), 31 (для двойных слов) или 63 (для учетверенных слов), то значение операнда приемник устанавливается равным 0.

PSUBB/PSUBW/PSUBD

- PSUBB приемник, источник
  PSUBW приемник, источник
  PSUBD приемник, источник
- OF F8 /r PSUBB rmmx1,rmmx2/m64
  66 OF F8 /r PSUBB rxmml, rxmm2/m128
  OF F9 /r . PSUBW rmmxl, rmmx2/m64
  66 OF F9 /r PSUBW rxmm1, rxmm2/m128
  OF FA /r PSUBD rmmxl, rmmx2/m64
  66 OF FA /r PSUBD rxmm1, rxmm2/ml28
- Вычитание упакованных байтов (слов, двойных слов).

Действие: команда вычитает из элементов источника элементы приемника размером байт (слово, двойное слово). При переполнении результат формируется в соответствии с принципом циклического переполнения. Результат помещается в операнд приемник.

PSUBQ

- PSUBQ приемник, источник
- OF FB /r PSUBQ rmmx1,rmmx2/m64
  66 OF FB /r PSUBQ rxmm1,rxmm2/m128
- Вычитание учетверенных слов.

Действие: выполняется вычитание 64-разрядных целых значений в источнике и приемнике. Исходя из типа источника, возможны две схемы умножения:

- если источник — MMX-регистр или 64-разрядная ячейка памяти, то из разрядов 0...63 приемника вычитаются разряды 0...63 источника и помещаются в разряды 0...63 приемника (MMX-регистра);
- если источник — XMM-регистр или 128-разрядная ячейка памяти, то из разрядов 0...63 приемника вычитаются разряды 0...63 источника и помещаются в разряды 0...63 приемника, а из разрядов 64...127 приемника вычитаются разряды 64...127 источника и помещаются в разряды 64...127.

В результате выполнения команды PSUBQ регистр EFLAGS не отражает факта возникновения ситуации переполнения или переноса. Когда результат умножения слишком большой, чтобы быть представленным в 64-разрядном элементе приемника, то он «заворачивается» (перенос игнорируется). Для обнаружения подобных ситуаций программное обеспечение должно использовать другие методы.

PSUBSB/PSUBSW

- PSUBSB приемник, источник
  PSUBSW приемник, источник
- 0F E8 /r PSUBSB rmmx1,rmmx2/m64
  66 0F E8 /r PSUBSB rxmm1, rxmm2/m128
  0F E9 /r PSUBSW rmmx1,rmmx2/m64
  66 0F E9 /r PSUBSW rxmml, rxmm2/m128
- Вычитание со знаковым насыщением упакованных байтов (слов).

Действие: команда вычитает элементы источника и приемника размером байт (слово) в зависимости от кода операции. Вычитание элементов производится с учетом их знака. При переполнении результат формируется в соответствии с принципом знакового насыщения:

- PSUBSB — 07fh для положительных чисел и 080h для отрицательных;
- PSUBSW — 07fffh для положительных чисел и 08000h для отрицательных.

Результат помещается в операнд приемник.

PSUBUSB/PSUBUSW

- PSUBUSB приемник, источник
  PSUBUSW приемник, источник
- 0F D8 /r PSUBUSB rmmx1, rmmx2/m64
  66 0F D8 /r PSUBUSB rxmml, rxmm2/m128
  0F D9 /r PSUBUSW rmmx1,rmmx2/m64
  66 0F D9 /r PSUBUSW rxmml, rxmm2/ml28
- Вычитание с беззнаковым насыщением упакованных байтов (слов).

Действие: команда вычитает без учета знака элементы операндов источник и приемник размером байт (слово) в зависимости от кода операции. При переполнении результат формируется в соответствии с принципом беззнакового насыщения:

- PSUBUSB — 00h для результатов вычитания меньших нуля;
- PSUBUSW — 0000h для результатов вычитания меньших нуля.

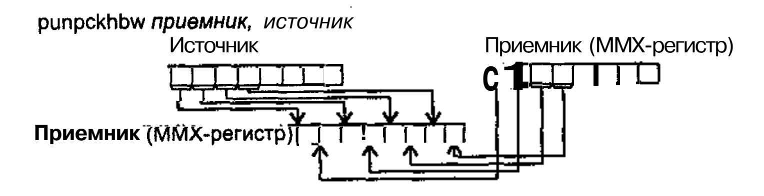
Результат помещается в операнд приемник.

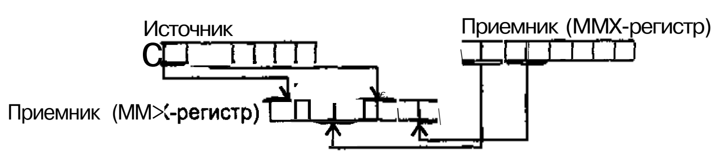
PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ

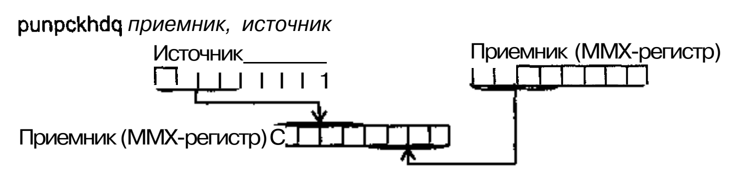
- PUNPCKHBW приемник, источник
  PUNPCKHWD приемник, источник
  PUNPCKHDQ приемник, источник
  PUNPCKHQDQ приемник, источник
- 0F 68 /r PUNPCKHBW rmmxl,rmmx2/m64
  66 0F 68 /r PUNPCKHBW rxmml, rxmm2/ml28
  0F 69 /r PUNPCKHWD rmmx1,rmmx2/m64
  66 0F 69 /r PUNPCKHWD rxmm1, rxmm2/m128
  0F 6A /r PUNPCKHDQ rmmx1,rmmx2/m64
  66 0F 6A /r PUNPCKHDQ rxmml, rxmm2/ml28
  66 0F 6D /r PUNPCKHQDQ rxmml, rxmm2/ml28
- Распаковка старших упакованных байтов (слов, двойных слов, учетверенных слов) в слова (двойные слова, учетверенные слова, сдвоенные учетверенные слова).

Действие: MMX-команды PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ производят размещение элементов из операндов источник и приемник согласно схем, представленных на рисунках (см. следующую страницу).

Результат помещается в операнд приемник. XMM-команды PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ производят размещение с чередованием элементов из операндов источник и приемник согласно представленной далее схеме:







- PUNPCKHBW:
  - приемник[7...0] ← приемник[71...64];
  - приемник[71...64] ← приемник[103...96];
  - приемник[15...8] ← источник[71...64];
  - приемник[79...72] ← источник[103...96];
  - приемник[23...16] ← приемник[79...72];
  - приемник[87...80] ← приемник[111...104];
  - приемник[31...24] ← источник[79...72];
  - приемник[95...88] ← источник[111...104];
  - приемник[39...32] ← приемник[87...80];
  - приемник[103...96] ← приемник[119...112];
  - приемник[47...40] ← источник[87...80];
  - приемник[111...104] ← источник[119...112];
  - приемник[55...48] ← приемник[95...88];
  - приемник[119...112] ← приемник[127...120];
  - приемник[63...56] ← источник[95...88];
  - приемник[127...120] ← источник[127...120].
- PUNPCKHWO:
  - приемник[15...0] ← приемник[79...64];
  - приемник[79...64] ← приемник[111...96];
  - приемник[31...16] ← источник[79...64];
  - приемник[95...80] ← источник[111...96];
  - приемник[47...32] ← приемник[95...80];

- □ приемник[111...96] ← приемник[127...112];
- □ приемник[63...48] ← источник[95...80];
- П приемник[127...112] ← источник[127...112].

PUNPCKHDQ:

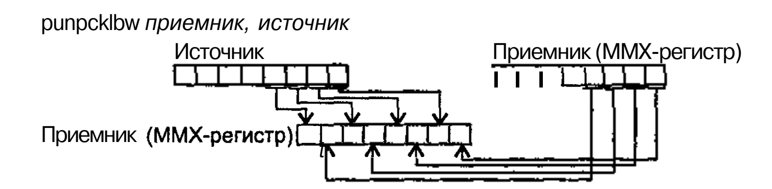
- D приемник[31...0] ← приемник[95...64];
- П приемник[95...64] ← приемник[127...96];
- D приемник[63...32] ← источник[95...64];
- D приемник[127...96] ← источник[127...96].

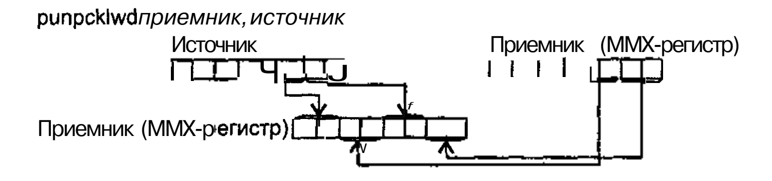
PUNPCKHQDQ:

- П приемник[63...0] ← приемник[127...64];
- П приемник[127...64] ← источник[127...64].

PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ

- PUNPCKLBW приемник, источник
  PUNPCKLWD приемник, источник
  PUNPCKLDQ приемник, источник
  PUNPCKLQDQ приемник, источник
- OF 60 /r PUNPCKLBW rmmx1,rmmx2/m64
  OF 61 /r PUNPCKLWD rmmx1,rmmx2/m64
  OF 62 /r PUNPCKLDQ rmmx1,rmmx2/m64
  66 OF 60 /r PUNPCKLBW rxmm1, rxmm2/m128
  66 OF 61 /r PUNPCKLWD rxmml, rxmm2/ml28
  66 OF 62 /r PUNPCKLDQ rxmml, rxmm2/m128
  66 OF 6C /r PUNPCKLQDQ rxmml, rxmm2/ml28







- Распаковка младших упакованных байтов (слов, двойных слов, учетверенных слов) в слова (двойные слова, учетверенные слова, сдвоенные учетверенные слова).

Действие: ММХ-команды PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ производят размещение элементов из операндов источник и приемник согласно схемам, представленным на рисунках (см. предыдущую страницу).

ХММ-команды PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ производят размещение с чередованием элементов из операндов источник и приемник согласно представленной далее схеме:

- PUNPCKLBW:
  - приемник[7...0] ← приемник[7...0];
  - приемник[71...64] ← приемник[39...32];
  - приемник[15...8] ← источник[7...0];
  - приемник[79...72] ← источник[39...32];
  - приемник[23...16] ← приемник[15...8];
  - приемник[87...80] ← приемник[47...40];
  - приемник[31...24] ← источник[15...8];
  - приемник[95...88] ← источник[47...40];
  - приемник[39...32] ← приемник[23...16];
  - приемник[103...96] ← приемник[55...48];
  - приемник[47...40] ← источник[23...16];
  - приемник[111...104] ← источник[55...48];
  - приемник[55...48] ← приемник[31...24];
  - приемник[119...112] ← приемник[63...56];
  - приемник[63...56] ← источник[31...24];
  - приемник[127...120] ← источник[63...56].
- PUNPCKLWD:
  - приемник[15...0] ← приемник[15...0];
  - приемник[79...64] ← приемник[47...32];
  - приемник[31...16] ← источник[15...0];
  - приемник[95...80] ← источник[47...32];
  - приемник[47...32] ← приемник[31...16];
  - приемник[111...96] ← приемник[63...48];
  - приемник[63...48] ← источник[31...16];
  - приемник[127...112] ← источник[63...48].
- PUNPCKLDQ:
  - приемник[31...0] ← приемник[31...0];
  - приемник[95...64] ← приемник[63...32];
  - приемник[63...32] ← источник[31...0];
  - приемник[127...96] ← источник[63...32].
- PUNPCKLQDQ:
  - приемник[63...0] ← приемник[63...0];
  - приемник[127...64] ← источник[63...0].

PXOR

- PXOR приемник, источник
- 0F EF/r PXOR rmmx1,rmmx2/m64
  66 0F EF /r PXOR rxmm1,rxmm2/m128
- Упакованное логическое исключающее ИЛИ.

Действие: команда производит поразрядную операцию логического исключающего ИЛИ над всеми битами операндов источник и приемник. Результат помещается в операнд приемник.

Команды блока XMM (SSE и SSE2)

ADDPD

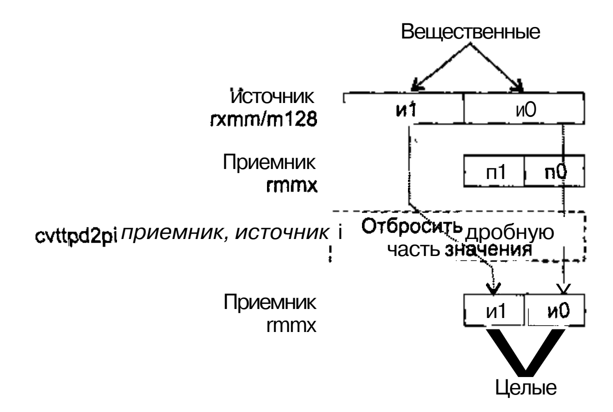
- ADDPD приемник, источник
- 66 0F 58 /r ADDPD rxmml, rxmm2/m128
- Сложение упакованных значений с плавающей точкой двойной точности.

Действие: пары упакованных значений с плавающей точкой двойной точности источника и приемника складываются (аналогично команде ADDPS), результат сложения сохраняется в соответствующих упакованных значениях с плавающей точкой двойной точности приемника.

ADDPS

- ADDPS приемник, источник
- 0F 58 /r ADDPS rxmm1,rxmm2/m128
- Сложение упакованных значений в формате XMM.

Действие: алгоритм работы команды показан на следующей схеме.



ADDSD

- ADDSD приемник, источник
- F2 0F 58 /r ADDSD rxmml, rxmm2/m64
- Сложение скалярных упакованных значений с плавающей точкой двойной точности.

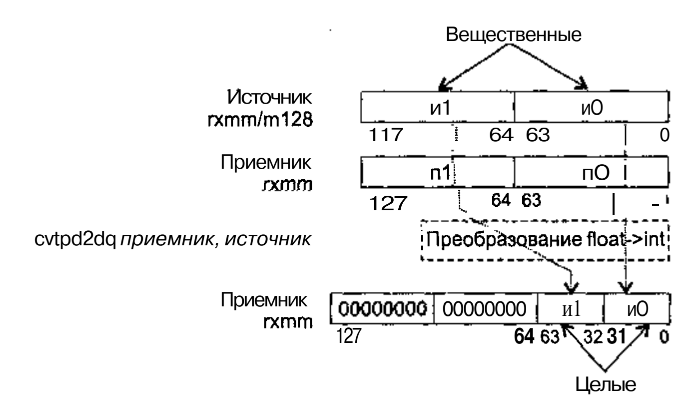
Действие: младшие упакованные значения с плавающей точкой двойной точности источника и приемника складываются (аналогично команде ADDSS), результат сложения сохраняется в младшем упакованным значении с плавающей точкой двойной точности приемника.

ADDSS

- ADDSS приемник, источник

- F3 0F 58 /r ADDSS rxmm1, rxmm2/m32
- Скалярное сложение значений в формате XMM.

Действие: алгоритм работы команды показан на следующей схеме:



ANDPD

- ANDPD приемник, источник
- 66 0F 54 /r ANDPD xmm1, xmm2/m128
- Поразрядное логическое И над упакованными значениями с плавающей точкой двойной точности.

Действие: команда выполняет операцию поразрядного логического И над двумя упакованными значениями с плавающей точкой двойной точности. Результат помещается в приемник.

ANDPS

- ANDPS приемник, источник
- 0F 54 /r ANDPS rxmm1,rxmm2/m128
- Поразрядное логическое И над каждой парой битов операндов.

ANDNPD

- ANDNPO приемник, источник
- 66 0F 55 /r ANDNPD xmm1, xmm2/m128
- Поразрядное логическое И-НЕ над упакованными значениями с плавающей точкой двойной точности.

Действие: команда выполняет операцию поразрядного логического И-НЕ над парами упакованных значений с плавающей точкой двойной точности в приемнике и источнике. Результат помещается в приемник.

ANDNPS

- ANDNPS приемник, источник
- 0F 55 /r ANDNPS rxmm1,rxmm2/m128
- Поразрядное логическое И-НЕ над упакованными значениями в формате XMM.

Действие: команда инвертирует биты операнда приемник и после инвертирования над каждой парой битов операндов приемник и источник выполняет операцию логического И.

CLFLUSH

- CLFLUSH адрес_байта

- 0F AE /7 CLFLUSH m8
- Сброс на диск строки кэша, содержащую адрес байта.

Действие: команда объявляет недействительной строку кэша, которая содержит линейный адрес байта, указанный операндом, на всех уровнях иерархии кэшей данных и команд процессора. Если на одном из уровней иерархии кэшей строка противоречит памяти (была изменена), то перед тем как быть объявленной недействительной, она записывается в память.

Возможность использования команды CLFLUSH на том или ином процессоре выясняется с помощью команды CPUID. Выровненный размер строки кэша, на который воздействует CLFLUSH, также определяется командой CPUID.

CMPPD

- CMPPD приемник, источник, условие
- 66 0F C2 /r i8 CMPPD xmm1, xmm2/m128, imm8
- Сравнение упакованных значений с плавающей точкой двойной точности.

Действие: команда сравнивает упакованные значения с плавающей точкой двойной точности в приемнике и источнике. Результат сравнения для каждой пары упакованных чисел представляется в виде маски: единичная маска ffffffffffffffffh — значения чисел равны, нулевая маска 0000000000000000h — значения не равны. Условие сравнения задается непосредственным операндом условие, первые 3 бита которого определяют тип сравнения. Остальные биты зарезервированы. Соответствие значений операнда условие условию сравнения:

- 0 — приемник - источник;
- 1 — приемник < источник;
- 2 — приемник < источник;
- 3 — приемник и/или источник — нечисло (NaN) или имеет неопределенный формат;
- 4 — приемник ≠ источник;
- 5 — ¬(приемник < источник);
- 6 — ¬(приемник < источник);
- 7 — упакованные значения приемника и источника — правильные значения с плавающей точкой двойной точности.

Для проверки остальных условий необходимо вначале поменять содержимое приемника и источника, а затем использовать команду CMPPD со следующими значениями операнда условие:

- 1 — приемник > источник;
- 2 — приемник > источник;
- 5 — ¬(приемник > источник);
- 6 — ¬(приемник > источник).

CMPPS

- CMPPS приемник, источник, условие
- 0F C2 /r ib CMPPS rxmm1, rxmm2/m128, i8
- Сравнение упакованных значений в формате XMM.

Действие: условие, в соответствии с которым производится сравнение каждой пары элементов операндов приемник и источник, задается явно в виде непосредственного операнда (табл. П.23). В результате сравнения в приемнике формируются единичные (если условие выполнено) или нулевые элементы (если условие не выполнено).

CMPSD

- CMPSD приемник, источник, условие
- F2 0F C2 /r i8 CMPSD xmm1, xmm2/m64, imm8
- Сравнение скалярных значений с плавающей точкой двойной точности.

Действие: команда сравнивает упакованные значения с плавающей точкой двойной точности в разрядах [0...63] приемника и источника. Формирование проверяемого условия и результата выполнения команды те же, что и для команды CMPPD.

Таблица П.23. Допустимые значения условия в команде CMPPS

Условие	Описание условия	Отношение	Эмуляция	Код маски **i8**	Результат, если операнд не **число (NaN)**	Исключение **#I,** если операнд QNaN/ SNaN
Eq	Равно	xmm1 = xmm2		000b	False	Нет
Lt	Меньше чем	xmm1 < xmm2		001b	False	Да
Le	Меньше чем или равно	xmm1 < xmm2		010b	False	Да
Gt	Больше чем	xmm1 > xmm2	Перестановка с сохранением, lt		False	Да
Ge	Больше чем или равно	xmm1 ≥ xmm2	Перестановка с сохранением, le		False	Да
Unord	Одно из чисел - NaN	xmm1 ? xmm2		011b	True	Нет
Neq	Не равно	xmm1 ≠ xmm2		100b	True	Нет
NJt	Не меньше чем	!(xmm1 < xmm2)		101b	True	Да
Nle	Не меньше чем или равно	!(xmm1 ≤ xmm2)		110b	True	Да
Ngt	Не больше чем	!(xmm1 > xmm2)	Перестановка с сохранением, nlt		True	Да
Nge	Не больше чем или равно	!(xmml>xmm2)	Перестановка с сохранением, nle		True	Да
Ord	Оба числа не NaN	xmm1 ? xmm2		111b	False	Нет

CMPSS

- CMPSS приемник, источник, условие
- F3 0F C2 /r ib   CMPSS rxmm1, rxmm2/m32, i8
- Скалярное сравнение значений в формате XMM.

Действие: для пары значений операндов приемник и источник выполняется сравнение, в результате которого формируются единичные (если условие выполнено) или нулевые элементы (если условие не выполнено). Значение источника может быть расположено в 32-разрядной ячейке памяти или в младшем двойном слове XMM-регистра. Значение приемника должно быть расположено в младшем двойном слове другого XMM-регистра. Возможные значения условий приведены в описании команды CMPPS (см. табл. П.23).

COMISD

- COMISD приемник, источник, условие
- 66 0F 2F /r COMISD xmm1, xmm2/m64
- Сравнение упорядоченных скалярных значений с плавающей точкой двойной точности и установка регистра EFLAGS.

Действие: упорядоченные скалярные значения с плавающей точкой двойной точности сравниваются в разрядах [0...63] приемника и источника. По результату сравнения устанавливаются флаги ZF, PF и CF в регистре EFLAGS:

- приемник > источник (ZF = O, PF = 0, CF = 0);
- приемник < источник (ZF = O, PF = O, CF = 1);
- приемник - источник (ZF = 1, PF - O, CF = 0);
- приемник и/или источник — нечисло (NaN) или имеют неопределенный формат (ZF = 1, PF = 1, CF-1).

Флаги OF, SF и AF устанавливаются в 0. В случае генерации немаскированного исключения с плавающей точкой регистр EFLAGS не модифицируется.

COMISS

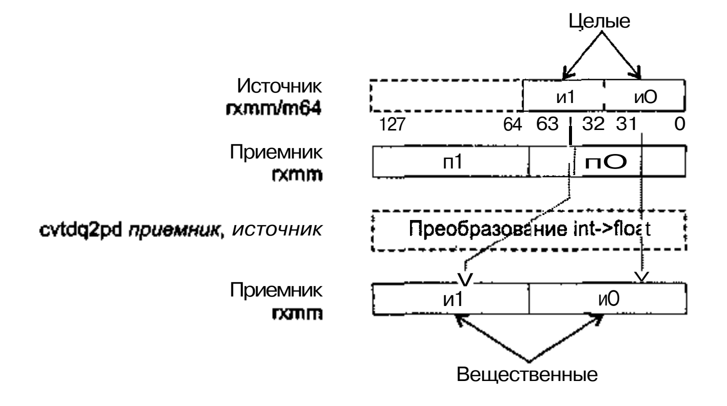
- COMISS приемник, источник
- 0F 2F /r COMISS rxmm1, rxmm2/m32
- Скалярное упорядоченное сравнение значений в формате XMM с установкой регистра EFLAGS.

Действие: команда сравнивает пару значений операндов приемник и источник, в результате устанавливаются флаги в регистре EFLAGS (табл. П.24). Значение источника может быть расположено в 32-разрядной ячейке памяти или младшем двойном слове XMM-регистра. Значение приемника должно быть расположено в младшем двойном слове другого XMM-регистра.

Таблица П.24. Допустимые значения флагов в регистре EFLAGS

Соотношение операндов	Значение флагов
Приемник > источник	OF-SF=AF=ZF-PF-CF-0
Приемник < источник	OF-SF-AF=ZF-PF=0; CF-1
Приемник = источник	OF-SF-AF-PF-CF-0; ZF=1
Приемник или источник = QNaN или SNaN	OF = SF - AF = 0; ZF - PF = CF = 1

При возникновении незамаскированных исключений значение EFLAGS не изменяется.



CVTDQ2PD

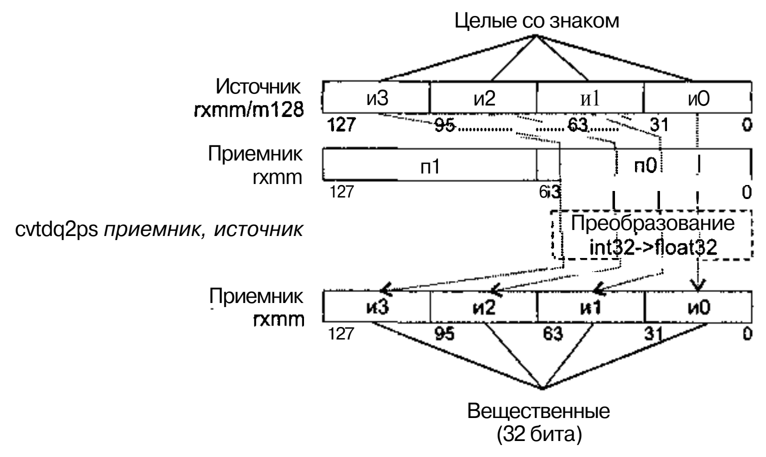
- CVTDQ2PD приемник, источник
- F3 0F E6 CVTDQ2PD rxmm1, rxmm2/m64
- Преобразование двух упакованных 32-разрядных целых в два упакованные значения плавающей точкой двойной точности.

  Действие: алгоритм работы команды показан на схеме (см. предыдущую страницу).

CVTDQ2PS

- CVTDQ2PS приемник, источник
- 0F 5B /r CVTDQ2PS rxmm1,rxmm2/m128
- Преобразование четырех упакованных 32-разрядных целых со знаком в четыреупакованных значения с плавающей точкой одинарной точности.

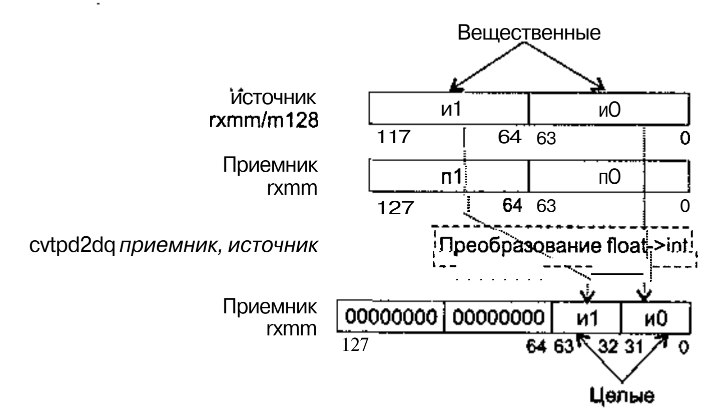
  Действие: алгоритм работы команды показан на следующей схеме.



В случае, когда не удается выполнить точное преобразование, значение округляется в соответствии с полем MXCSR.RC.

CVTPD2DQ

- CVTPD2DQ приемник, источник
- F2 0F E6 CVTPD2DQ rxmm1, rxmm2/ml28
- Преобразование двух упакованных значений с плавающей точкой двойной точности в два упакованных 32-разрядных целых.



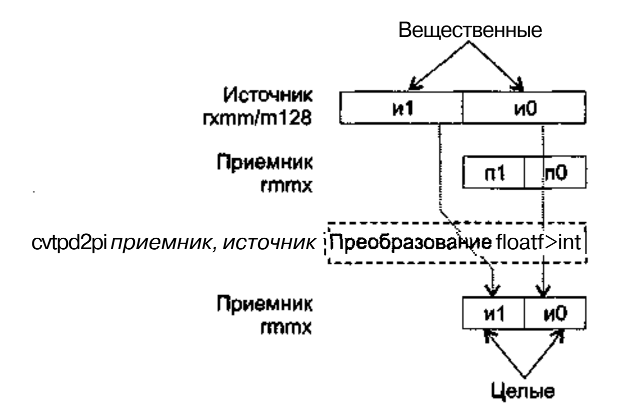
Действие: алгоритм работы команды показан на следующей схеме.

В случае, когда не удается выполнить точное преобразование, значение округляется в соответствии с полем MXCSR.RC. Если преобразованный результат больше чем максимально возможное целочисленное 32-разрядное значение, то возвращается значение 80000000h.

CVTPD2PI

- CVTPD2PI приемник, источник
- 66 0F 2D /r CVTPD2PI rmmx, rxmm/m128
- Преобразование двух упакованных значений с плавающей точкой двойной точности в два упакованных 32-разрядных целых.

Действие: алгоритм работы команды показан на следующей схеме.

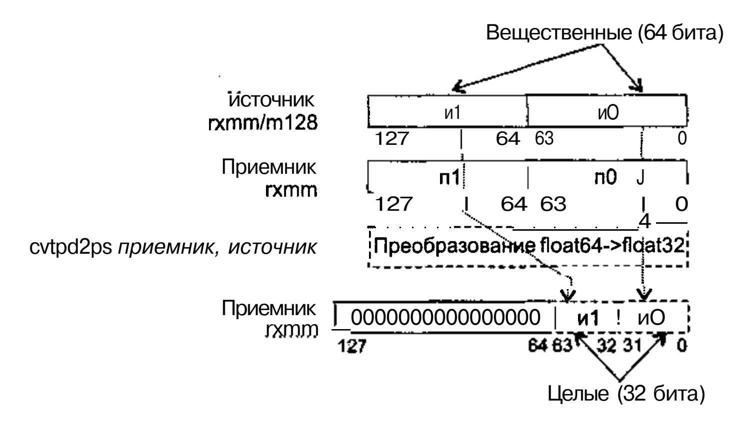


В случае, когда не удается выполнить точное преобразование, значение округляется в соответствии с полем MXCSR.RC. Если преобразованный результат больше, чем максимально возможное целочисленное 32-разрядное значение, то возвращается значение 80000000h.

CVTPD2PS

- CVTPD2PS приемник, источник
- 66 0F 5A /r CVTPD2PS rxmm1, rxmm2/m128
- Преобразование двух упакованных значений с плавающей точкой двойной точности в два упакованных значения с плавающей точкой одинарной точности.

Действие: алгоритм работы команды показан на следующей схеме.

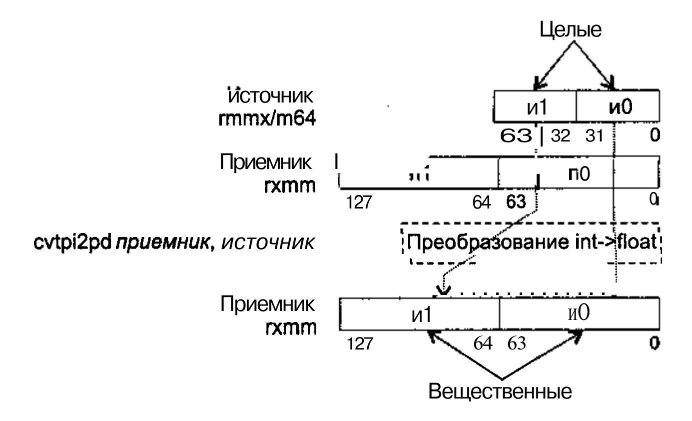


В случае, когда не удается выполнить точное преобразование, значение округляется в соответствии с полем MXCSR.RC.

CVTPI2PD

- CVTPI2PD приемник, источник
- 66 OF 2A/r CVTPI2PD rxmm, rmmx/m64
- Преобразование двух упакованных 32-разрядных целых в два упакованных значения с плавающей точкой двойной точности.

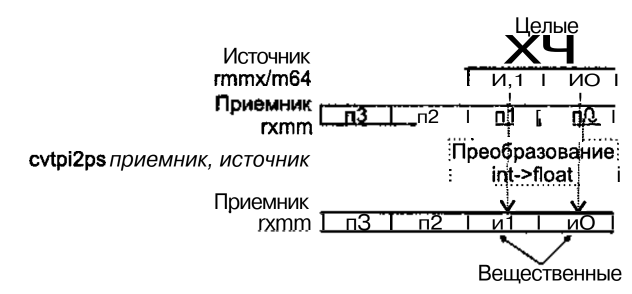
Действие: алгоритм работы команды показан на следующей схеме.



CVTPI2PS

- CVTPI2PS приемник, источник
- OF 2A /r CVTPI2PS rxmm,rmmx/m64
- Преобразование двух упакованных 32-разрядных целых в два упакованных вещественных значения.

Действие: алгоритм работы команды показан на следующей схеме.

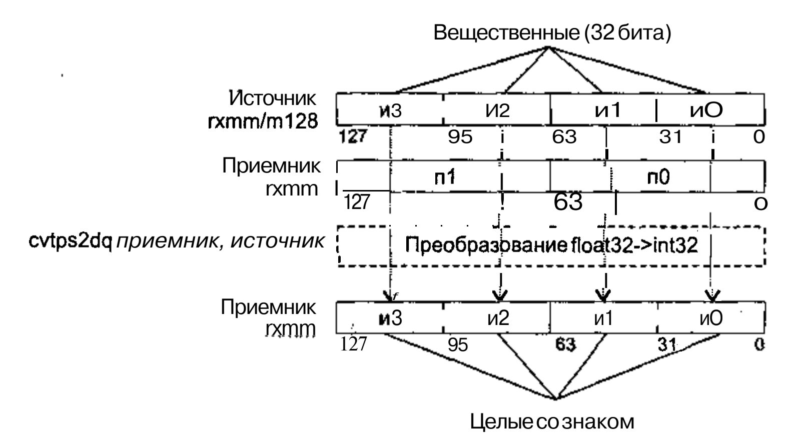


В случае, когда не удается выполнить точное преобразование, результат округляется в соответствии с полем MXCSR.RC.

CVTPS2DQ

- CVTPS2DQ приемник, источник
- 66 OF 5B /r CVTPS2DQ rxmm1, rxmm2/m128
- Преобразование четырех упакованных значений с плавающей точкой одинарной точности в четыре упакованных 32-разрядных целых со знаком.

Действие: алгоритм работы команды показан на следующей схеме.

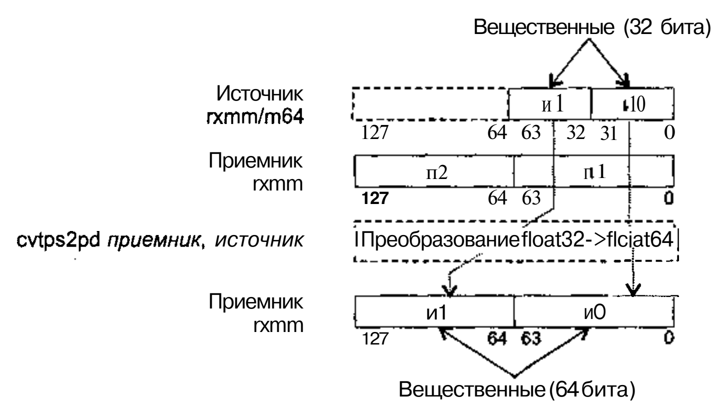


В случае, когда не удастся выполнить точное преобразование, значение округляется в соответствии с полем MXCSR.RC. Если преобразованный результат больше, чем максимально возможное целочисленное 32-разрядное значение, то возвращается значение 80000000h.

CVTPS2PD

- CVTPS2PD приемник, источник
- 0F 5A /r CVTPS2PD rxmm1, rxmm2/m64
- Преобразование двух упакованных значений с плавающей точкой одинарной точности в два упакованных значения с плавающей точкой двойной точности.

Действие: алгоритм работы команды показан на следующей схеме.

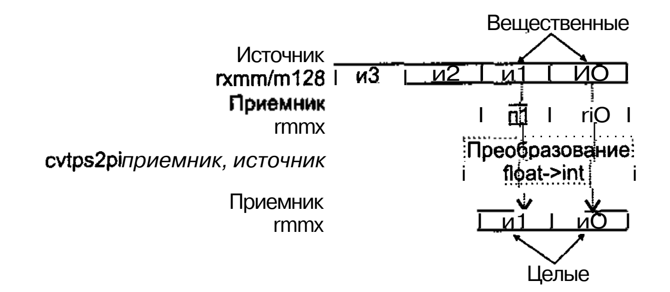


CVTPS2PI

- CVTPS2PI приемник, источник
- 0F 2D /r CVTPS2PI rmmx, rxmm/m128
- Преобразование двух вещественных целых в два упакованных 32-разрядных целых.

Действие: алгоритм работы команды показан на схеме (см. следующую страницу).

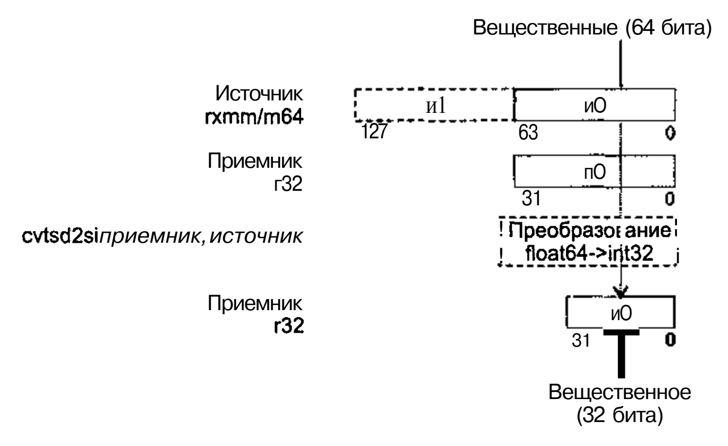
Если преобразованный результат больше, чем максимально возможное целочисленное 32-разрядное значение, то возвращается значение 80000000h. В случае, когда не удается выполнить точное преобразование, значение округляется в соответствии с полем MXCSR.RC.



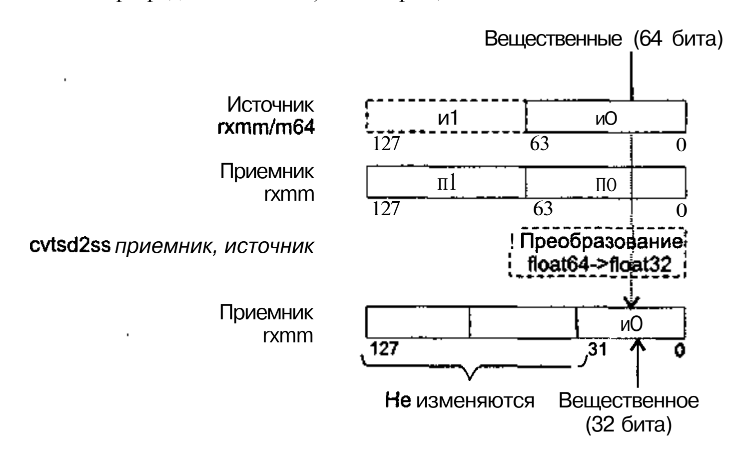
CVTSD2SI

- CVTSD2SI приемник, источник
- F2 0F 2D /r CVTSD2SI r32, rxmm/m64
- Преобразование скалярного значения с плавающей точкой двойной точности в 32-разрядное целое.

Действие: алгоритм работы команды показан на следующей схеме.



В случае, когда не удается выполнить точное преобразование, значение округляется в соответствии с полем MXCSR.RC. Если преобразованный результат больше, чем максимально возможное целочисленное 32-разрядное значение, то возвращается значение 80000000h.



CVTSD2SS

- CVTSD2SS приемник, источник
- F2 0F 5A /r CVTSD2SS rxmm1, rxmm2/m64
- Преобразование скалярного значения с плавающей точкой двойной точности в скалярное значение с плавающей точкой одинарной точности.

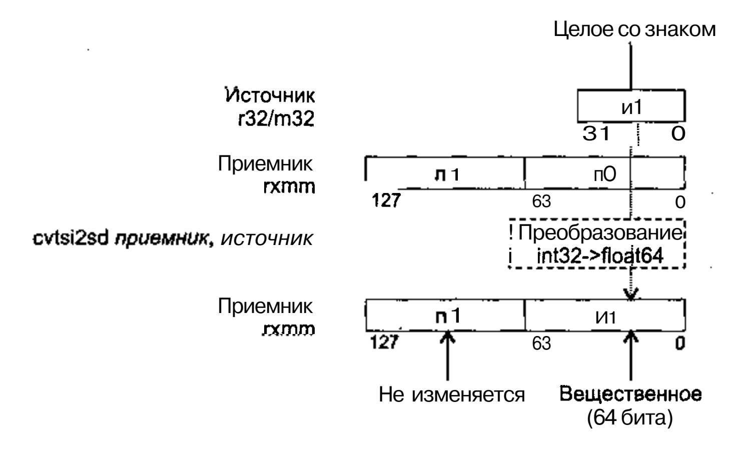
Действие: алгоритм работы команды показан на схеме (см. предыдущую страицу).

В случае, когда не удается выполнить точное преобразование, значение округляется в соответствии с полем MXCSR.RC.

CVTSI2SD

- CVTSI2SD приемник, источник
- F2 0F 2A/r CVTSI2SD rxmm, r/m32
- Преобразование 32-разрядного целого значения со знаком в упакованное значение с плавающей точкой двойной точности.

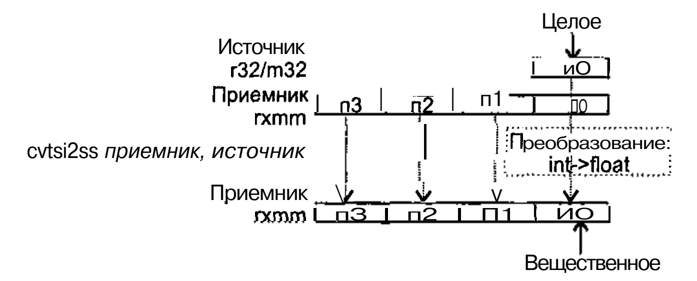
Действие: алгоритм работы команды показан на следующей схеме.



CVTSI2SS

- CVTSI2SS приемник, источник
- F3 0F 2A/r CVTSI2SS rxmm, r/m32
- Скалярное преобразование знакового 32-разрядного целого в вещественное значение.

Действие: алгоритм работы команды показан на следующей схеме.

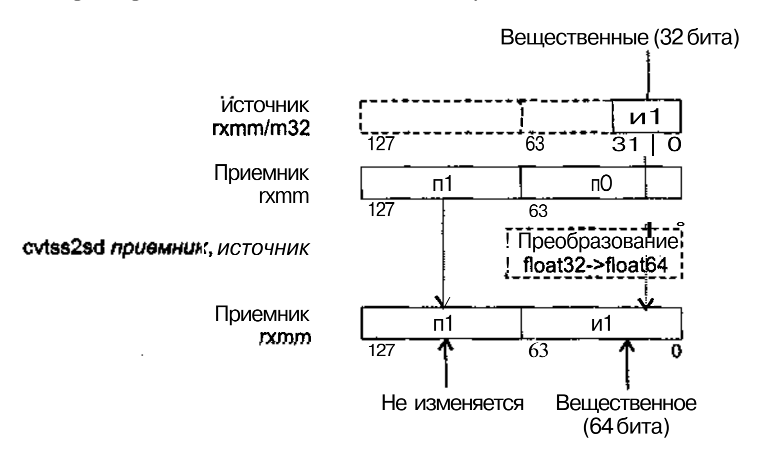


В случае, когда не удается выполнить точное преобразование, значение округляется в соответствии с полем MXCSR.RC.

CVTSS2SD

- CVTSS2SD приемник, источник
- F3 0F 5A /r CVTSS2SD rxmm1,rxmm2/m32
- Преобразование скалярного значения с плавающей точкой одинарной точности в скалярное значение с плавающей точкой двойной точности.

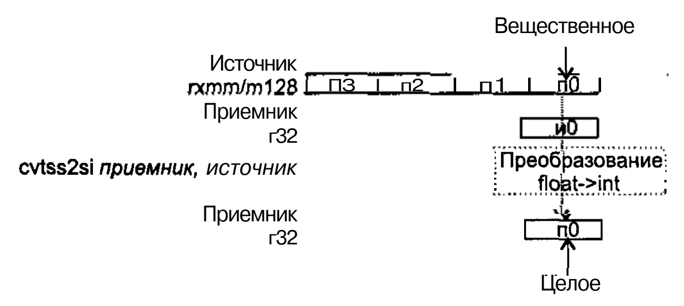
Действие: алгоритм работы команды показан на следующей схеме.



CVTSS2SI

- CVTSS2SI приемник, источник
- F3,0F,2D,/r CVTSS2SI r32,rxmm/m32
- Скалярное преобразование вещественного целого в 32-разрядное знаковое целое.

Действие: значение источника хранится в младшем двойном слове ХММ-регистра или в 32-разрядной ячейке памяти. Приемник — один из 32-разрядных регистров. Алгоритм работы команды показан на следующей схеме.

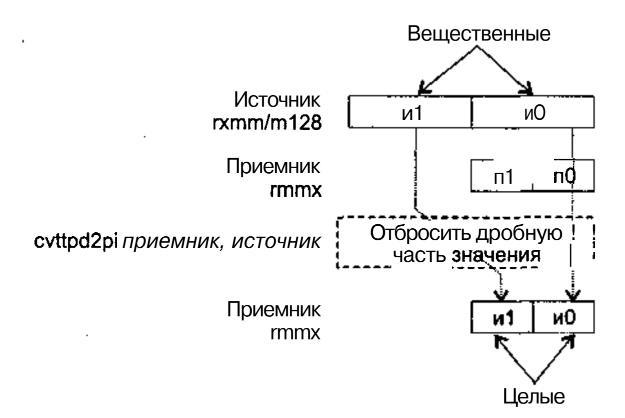


Если преобразованный результат больше, чем максимально возможное целочисленное 32-разрядное значение, то возвращается значение 80000000h. В случае, когда не удается выполнить точное преобразование, значение округляется в соответствии с полем MXCSR.RC.

CVTTPD2PI

- CVTTPD2PI приемник, источник
- 66 0F 2C /r CVTTPD2PI mm, xmm/m128
- Преобразование (путем отбрасывания дробной части) двух упакованных значений с плавающей точкой двойной точности в два упакованных 32-разрядных целых значения.

Действие: алгоритм работы команды показан на следующей схеме.

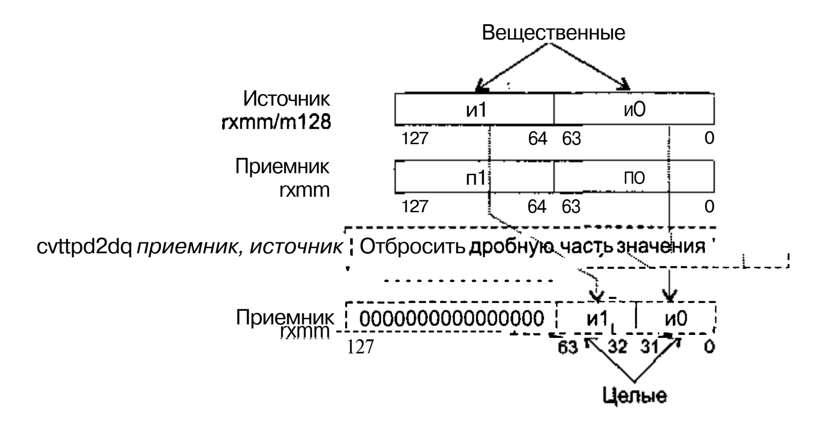


Если преобразованный результат больше, чем максимально возможное целочисленное 32разрядное значение, то будет возвращено значение 80000000h.

CVTTPD2DQ

- CVTTPD2DQ приемник, источник
- 66 0F E6 CVTTPD2DQ rxmm1, rxmm2/m128
- Преобразование усечением двух упакованных значений с плавающей точкой двойной точности в два упакованных 32-разрядных целых.

  Действие: алгоритм работы команды показан на следующей схеме.

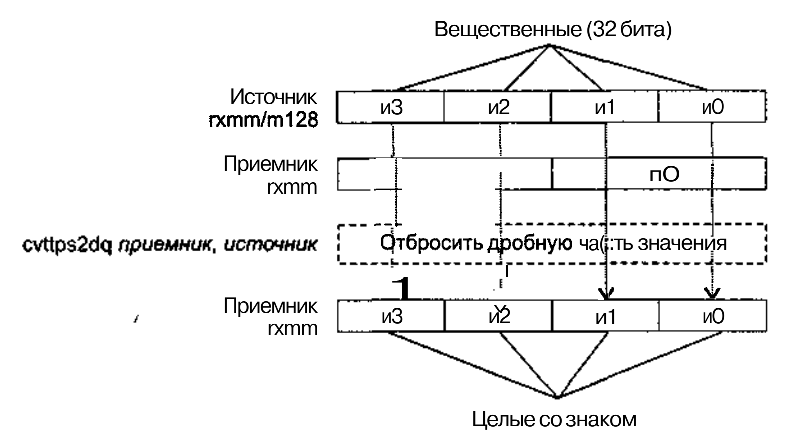


В случае, когда не удается выполнить точное преобразование, значение округляется в сторону нуля. Если преобразованный результат больше, чем максимально возможное целочисленное 32-разрядное значение, то возвращается значение 80000000h.

CVTTPS2DQ

- CVTTPS2DQ приемник, источник
- F3 0F 5B /r CVTTPS2DQ rxmm1, rxmm2/m128
- Преобразование (путем отбрасывания дробной части) четырех упакованных значений с плавающей точкой одинарной точности в четыре упакованных 32-разрядных целых со знаком.

Действие: алгоритм работы команды показан на следующей схеме.

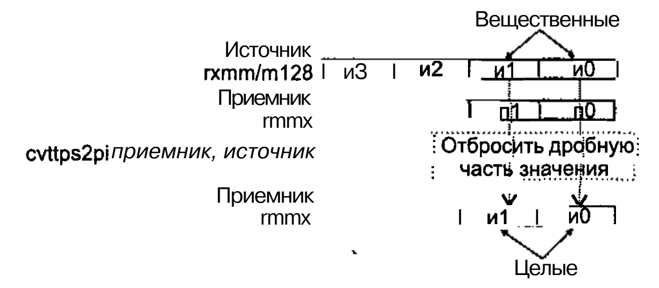


В случае, когда не удается выполнить точное преобразование, значение округляется в сторону нуля. Если преобразованный результат больше, чем максимально возможное целочисленное-32-разрядное значение, то будет возвращено значение 80000000h.

CVTTPS2PI

- CVTTPS2PI приемник, источник
- 0F 2C /r CVTTPS2PI rmmx, rxmm/m64
- Преобразование (путем отбрасывания дробной части) двух вещественных целых в два упакованных 32-разрядных целых значения.

Действие: алгоритм работы команды показан на следующей схеме.



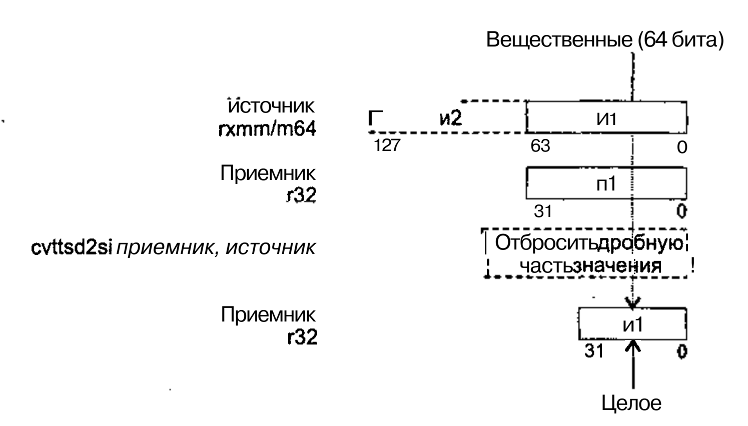
Если преобразованный результат больше, чем максимально возможное целочисленное 32-разрядное значение, то будет возвращено значение 80000000h.

CVTTSD2SI

- CVTTSD2SI приемник, источник
- F2 0F 2C /r CVTTSD2SI r32, rxmm/m64
- Преобразование (путем отбрасывания дробной части) скалярного значения с плавающей точкой двойной точности в 32-разрядное целое.

Действие: алгоритм работы команды показан на схеме (см. следующую страницу).

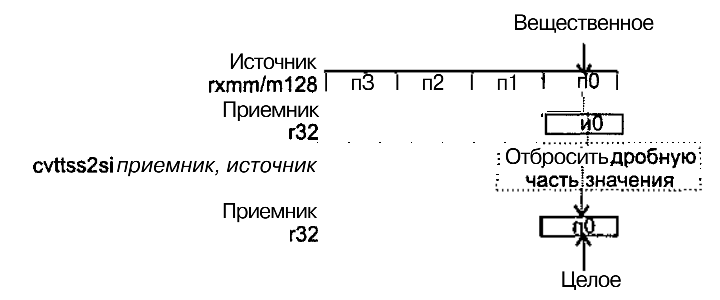
В случае, когда не удается выполнить точное преобразование, значение округляется в сторону нуля. Если преобразованный результат больше, чем максимально возможное целочисленное-32-разрядное значение, то будет возвращено значение 80000000h.



CVTTSS2SI

- CVTTSS2SI приемник, источник
- F3 0F 2C /r CVTTSS2SI r32, rxmm/m32
- Скалярное преобразование (путем отбрасывания дробной части) вещественного целого в знаковое целое.

Действие: значение источника хранится в младшем двойном слове XMM-регистра или в 128-разрядной ячейке памяти. Приемник — один из 32-разрядных регистров. Алгоритм работы команды показан на следующей схеме.



Если преобразованный результат больше, чем максимально возможное целочисленное 32-разрядное значение, то будет возвращено значение 80000000h.

DIVPD

- DIVPD приемник, источник
- 66 0F 5E /r DIVPD xmm1, xmm2/m128
- Деление упакованных значений с плавающей точкой двойной точности.

Действие: команда делит пары упакованных значений с плавающей точкой двойной точности источника и приемника по следующей схеме: разряды 0...63 приемника делятся на разряды 0...63 источника и результат помещается в разряды 0...63 приемника, разряды 64...127 приемника делятся на разряды 64...127 источника и результат помещается в разряды 64...127 приемника.

DIVPS

- DIVPS приемник, источник
- 0F 5E /r DIVPS rxmm1,rxmm2/m128

• Деление упакованных значений в формате XMM.

Действие: Алгоритм работы команды показан на следующей схеме.

divps *приемник, источник*

Приемник rxmm	п3	п2	п1	п0
	/	/	/	/
Источник rxmm/m128	и3	и2	и1	и0
	=	=	=	=
Приемник rxmm	п3/и3	п2/и2	п1/и1	п0/и0

DIVSD

• DIVSD приемник, источник

• F2 0F 5E /r DIVSD rxmm1, rxmm2/m64

• Деление скалярных упакованных значений с плавающей точкой двойной точности.

Действие: команда делит младшие упакованные значения с плавающей точкой двойной точности источника и приемника по следующей схеме: разряды 0...63 приемника делятся на разряды 0...63 источника и результат помещается в разряды 0...63 приемника, разряды 64...127 приемника не изменяются.

DIVSS

• DIVSS приемник, источник

• F3 0F 5E /r DIVSS rxmm1, rxmm2/m32

• Скалярное деление значений в формате XMM.

Действие: Алгоритм работы команды показан на следующей схеме.

divss *приемник, источник*

Приемник rxmm	п3	п2	п1	п0
				/
Источник rxmm/m32	и3	и2	и1	и0
				=
Приемник rxmm	п3	п2	п1	п0/и0

FXRSTOR

• FXRSTOR источник

• 0F AE /1 FXRSTOR m512byte

• Восстановление состояния FPU, MMX и XMM, а также регистра MXCSR.

Действие: восстановление состояния сопроцессора (без проверки незамаскированных исключений), целочисленного и потокового MMX-расширений из 512-разрядной области памяти. Алгоритм работы команды показан на схеме (см. следующую страницу).

FXSAVE

• FXSAVE приемник

• 0F AE /0 FXSAVE m512byte

• Сохранение состояния FPU, MMX и XMM, а также регистра MXCSR.

Действие: сохранение состояния сопроцессора, целочисленного и потокового MMX-расширений в 512-разрядной области памяти (см. схему в описании команды FXRSTOR).

	15 14	13 12	11 10 9 8	7 6	5 4	3 2	1 0	
	Резерв	cs	ip	FOP	FTW	FSW	FCW	0
	Резерв		mxcsr	Резерв	ds	dp		16
	Резерв			st0/rmmx0				32
	Резерв			st1/rmmx1				48
	Резерв			st2/rmmx2				64
	Резерв			st3/rmmx3				80
	Резерв			st4/rmmx4				96
	Резерв			st5/rmmx5				112
	Резерв			st6/rmmx6				128
	Резерв			st7/rmmx7				144
	rxmm0							160
	rxmm1							176
	rxmm2							192
	rxmm3							208
	rxmm4							224
	rxmm5							240
	rxmm6							256
	rxmm7							272
	Резерв							288
	Резерв							...
512	Резерв							496

LDMXCSR

- LDMXCSR источник
- 0F AE /2 **LDMXCSR** m32
- Загрузка регистра состояния/управления MXCSR из 32-разрядной ячейки памяти.

Действие: Алгоритм работы команды показан на следующей схеме.

Регистр состояния/управления mxcsr

резерв	fz	rc	rc	pm	um	om	zm	dm	im	–	pe	ue	oe	ze	de	ie

31 15 10 5 O

Замечание: по умолчанию регистр MXCSR загружается значением 1f80.

LFENCE

- LFENCE адрес_байта
- 0F AE /5 LFENCE
- Упорядочение операций загрузки.

Действие: выполняется упорядочение (сериализация) команд загрузки из памяти, которые были инициированы перед вызовом команды LFENCE. Эта операция гарантирует, что каждая команда загрузки, за которой в программе следует команда LFENCE, будет глобально видима перед любой другой командой загрузки, за которой следует команда LFENCE. Команда LFENCE упорядочивается относительно команд загрузки, других команд LFENCE, команд MFENCE и любых команд сериализации (вроде команды CPUID). Она не упорядочивается относительно команд сохранения в памяти или команды SFENCE.

MASKMOVDQU

- MASKMOVDQU источник, маска
- 66 0F F7 /r MASKMOVDQU rxmm1, rxmm2
- Выборочная запись байтов из источника в память с использованием байтовой маски в приемнике.

Действие: команда сохраняет выбранные байты операнда источник в 128-разрядной ячейке памяти. Операнд маска определяет, какие байты источника сохраняются в памяти. Местоположение первого байта указанной операндом приемник ячейки памяти, в которой сохраняются байты, определяются парой DS:DI/EDI. Старший значащий бит каждого байта операнда маска определяет, будет ли сохранен в приемнике соответствующий байт источника:

- О — байт не сохраняется;
- 1 — байт сохраняется.

Команда MASKMOVEDQU генерирует процессору указание не задействовать кэш (non-temporal hint). Это указание реализуется методом кэширования WC (Write Combining — память с комбинированной записью). Поскольку в методе WC применяется слабо упорядоченная модель совместимости памяти (weakly-ordered memory consistency model), то команды SFENCE или MFENCE, реализующие операции сериализации, необходимо использовать совместно с командами MASKMOVEDQU. Для многопроцессорной конфигурации это особенно важно, так как в различных процессорах могут применяться различные типы памяти для чтения/записи ячейки приемника.

MAXPD

- MAXPD приемник, источник
- 66 0F 5F /r MAXPD rxmm1, rxmm2/ml28
- Возврат максимальных упакованных значений с плавающей точкой двойной точности.

Действие: команда сравнивает упакованные значения с плавающей точкой двойной точности в источнике и приемнике, находит максимальное и заменяет им соответствующее упакованное значение в приемнике. Если в источнике находится значение SNaN (но не QNaN), то оно помещается в приемник. Если только одно значение в приемнике или источнике — нечисло (SNaN или QNaN), то в приемник помещается содержимое источника, которое может быть либо нечислом, либо правильным значением с плавающей точкой.

MAXPS

- MAXPS приемник, источник
- OF,5F,/r MAXPS rxmm1, rxmm2/ml28
- Возврат максимального из упакованных значений в формате XMM.

Действие: команда извлекает максимальные значения в каждой из четырех пар вещественных чисел в коротком формате. При этом происходит сравнение значений соответствующих элементов источника и приемника, по результатам которого выполняются следующие действия:

- если элемент приемника или источника является сигнальным нечислом (SNaN), то в элемент приемника помещается значение источника;
- если элемент источника больше элемента приемника, то в элемент приемника помещается элемент источника.

В остальных случаях значения источника и приемника не меняются.

MAXSD

- MAX5D приемник, источник
- F2 OF 5F/r MAXSD rxmml,rxmm2/m64
- Возврат максимального скалярного значения с плавающей точкой двойной точности.

Действие: команда сравнивает значения с плавающей точкой двойной точности в разрядах 0...63 источника и приемника, находит максимальное из них и заменяет им значение в разрядах 0...63 приемника. Если значение в источнике - SNaN (но не QNaN), то оно помещается в приемник. Если только одно значение в приемнике или источнике - нечисло (SNaN или QNaN), то в приемник помещается содержимое источника, которое может быть либо нечислом, либо правильным значением числа с плавающей точкой. Значение в разрядах 64...127 приемника не меняется.

MAXSS

- MAXSS приемник, источник
- F3 0F 5F /r MAXSS rxmm1, rxmm2/m32
- Скалярный возврат максимального значения в формате XMM.

Действие: команда извлекает максимальное из двух вещественных чисел в коротком формате. При этом происходит сравнение значений младшей пары элементов источника и приемника, по результатам которого выполняются действия, аналогичные рассмотренным в описании команды MAXPS. Старшие три элемента источника и приемника не меняются.

MFENCE

- MFENCE
- 0F AE /6 . MFENCE
- Упорядочение операций загрузки и сохранения.

Действие: команда выполняет упорядочение (сериализацию) команд загрузки из памяти и сохранения в памяти, которые были инициированы перед вызовом команды MFENCE. Эта операция гарантирует, что каждая команда загрузки и сохранения, за которой в программе следует команда MFENCE, будет глобально видима перед любой другой командой загрузки и сохранения, за которой следует команда MFENCE. Команда MFENCE упорядочивается относительно команд загрузки и сохранения, других команд LFENCE, MFENCE, SFENCE и любых команд сериализации (вроде команды CPUID).

MINPD

- MINPD приемник, источник
- 66 0F 5D /r MINPD xmm1, xmm2/m128
- Возврат минимальных упакованных значений с плавающей точкой двойной точности.

Действие: команда сравнивает упакованные значения с плавающей точкой двойной точности в источнике и приемнике, обнаруживает минимальное и заменяет им соответствующее упакованное значение в приемнике. Если значение в источнике — SNaN (но не QNaN), то оно помещается в приемник. Если только одно значение в приемнике или источнике — нечисло (SNaN или QNaN), то в приемник помещается содержимое источника, которое может быть либо нечислом, либо правильным значением с плавающей точкой.

MINPS

- MINPS приемник, источник
- 0F 5D /r MINPS rxmm1, rxmm2/m128
- Возврат минимального упакованного значения в формате XMM.

Действие: команда извлекает минимальные значения в каждой из четырех пар вещественных чисел в коротком формате. При этом происходит сравнение значений соответствующих элементов источника и приемника, по результатам которого выполняются действия, аналогичные рассмотренным в описании команды MAXPS.

MINSD

- MINSD приемник, источник
- F2 0F 5D /r MINSD xmml, xmm2/m64
- Возврат минимального скалярного значения с плавающей точкой двойной точности.

Действие: команда сравнивает значения с плавающей точкой двойной точности в разрядах 0...63 источника и приемника, обнаруживает минимальное и заменяет им значение в разрядах 0...63 приемника. Если значение в источнике — SNaN (но не QNaN), то оно помещается в приемник. Если только одно значение в приемнике или источнике — нечисло (SNaN или QNaN), то в прием-

ник помещается содержимое источника, которое может быть либо нечислом, либо правильным значением с плавающей точкой. Значение в разрядах 64...127 приемника не меняется.

MIMSS

- MINSS приемник, источник
- F3 0F 5D /r MINSS rxmm1, rxmm2/m32
- Скалярный возврат минимальногозначения в формате XMM.

Действие: команда извлекает минимальное из двух вещественных чисел в коротком формате.Приэтомпроисходитсравнениезначениймладшейпарыэлементовисточникаиприемника,по результатам которого выполняются действия, аналогичные рассмотренным в описании команды MAXPS. Старшие три элемента источника и приемника не меняются.

MOVAPD

- MOVAPD приемник, источник
- 0F 28 /r MOVAPS rxmm1,rxmm2/m128
  0F 29 /r MOVAPS rxmm2/m128,rxmm1
- Перемещение упакованныхвыровненных значений с плавающей точкой двойной точности.

Действие: команда перемещает два двойных учетверенных слова (содержащих два упакованных значения с плавающей точкой двойной точности) из источника в приемник. Операнд в памяти должен быть выровнен по 16-разрядной границе.

MOVAPS

- MOVAPS приемник, источник
- 0F 28 /r MOVAPS rxmm1, xmm2/m128
  0F 29 /r MOVAPS rxmm2/ml28, rxmm1
- Перемещение 128 выровненных битов источника в соответствующие биты приемника.

MOVD

- MOVD приемник, источник
- См. описание команды MOVD в разделе «Команды блокаMMX».

MOVDQA

- MOVDQA приемник, источник
- 66 0F 6F /r MOVDQA rxmm1, rxmm2/ml28
  66 0F 7F /r MOVDQA rxmm2/ml28, rxmml
- Перемещение 128 выровненных битов из источника в приемник.

Действие: команда перемещает содержимое источника в приемник. Операнд в памяти должен быть выровнен по 16-разрядной границе.

MOVDQU

- MOVDQU приемник, источник
- F3 0F 6F /r MOVDQU xmm1, xmm2/m128
  F3 0F 7F /r MOVDQU xmm2/ml28, xmm1
- Перемещение 128 невыровненных битов из источника в приемник.

Действие: команда перемещает содержимое источника в приемник.

MOVDQ2Q

- MOVDQ2Q приемник, источник
- F2 0F D6 MOVDQ2Q rmmx, rxmm

- Перемещение младшего учетверенного слова XMM-регистра в MMX-регистр.

Действие: команда перемещает разряды 0...63 источника в приемник.

MOVHLPS

- MOVHLPS приемник, источник
- 0F 12 /r MOVHLPS rxmm1, rxmm2
- Копирование содержимого старшей половины XMM-регистра (источника) в младшую половину другого XMM-регистра (приемника).

Действие: команда перемещает разряды 64...127 источника в разряды 0...63 приемника. Разряды 64...127 приемника не меняются.

MOVHPD

- MOVHPD приемник, источник
- 66 0F 16 /r MOVHPD rxmm, m64
  66 0F 17 /r MOVHPD m64, rxmm
- Перемещение старшего упакованного значения с плавающей точкой двойной точности.

Действие: команда перемещает учетверенное слово (содержащее упакованное значение с плавающей точкой двойной точности) из источника в приемник. Источник и приемник могут быть либо XMM-регистром, либо 64-разрядной ячейкой памяти (но не одновременно). Для регистрового операнда перемещению подвергается старшее учетверенное слово (разряды 64...127). Младшее учетверенное слово XMM-регистра (разряды 0...63) не меняется.

MOVHPS

- MOVHPS приемник, источник
- 0F 16 /r MOVHPS xmm, m64
  0F 17 /r MOVHPS m64, xmm
- Перемещение верхних упакованных значений в формате XMM из источника в приемник.

Действие: команда имеет несколько вариантов действий:

- если источник — 64-разрядный операнд в памяти, то команда MOVHPS перемещает его содержимое в старшую половину приемника, представляющего собой XMM-регистр;
- если источник — XMM-регистр, то команда MOVHPS перемещает содержимое его старшей половины в приемник, который представляет собой 64-разрядный операнд в памяти.

MOVLHPS

- MOVLHPS приемник, источник
- 0F 16 /r MOVLHPS xmm1, xmm2
- Перемещение нижних упакованных значений в формате XMM в верхние.

Действие: команда копирует содержимое младшей половины XMM-регистра (источника) в старшую половину другого XMM-регистра (приемника). После операции изменяется только содержимое старшей половины приемника.

MOVLPD

- MOVLPD приемник, источник
- 66 0F 12 /r MOVLPD rxmm, m64
  66 0F 13 /r MOVLPD m64, rxmm
- Перемещение младшего упакованного значения с плавающей точкой двойной точности.

Действие: команда перемещает учетверенное слово (содержащее упакованное значение с плавающей точкой двойной точности) из источника в приемник. Источник и приемник могут быть либо XMM-регистром, либо 64-разрядной ячейкой памяти (но не одновременно). Для регистрового

операнда перемещению подвергается младшее учетверенное слово (разряды 0...63). Старшее учетверенное слово ХММ-регистра (разряды 64...127) не меняется.

MOVLPS

- MOVLPS приемник, источник
- 0F 12 /r MOVLPS rxmm, m64
  0F 13 /r MOVLPS m64, rxmm
- Перемещение невыровненных нижних упакованных значений в формате ХММ.

Действие: команда копирует содержимое младшей половины ХММ-регистра в 64-разрядную ячейку памяти или из нее:

- если источник — 64-разрядный операнд в памяти, то его содержимое перемещается в младшую половину приемника, представляющего собой ХММ-регистр;
- если источник — ХММ-регистр, то содержимое его младшей половины перемещается в приемник, который представляет собой 64-разрядный операнд в памяти.

MOVMSKPD

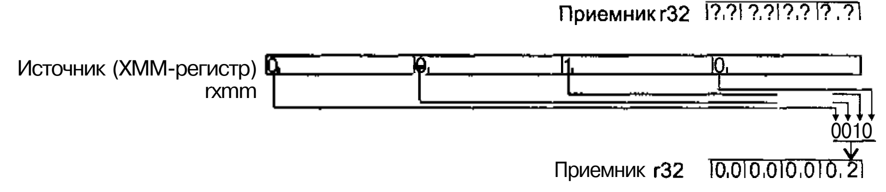
- MOVMSKPD приемник, источник
- 66 0F 50 /r MOVMSKPD r32, rxmm
- Извлечение 2-разрядной знаковой маски упакованных значений с плавающей точкой двойной точности.

Действие: команда извлекает знаковые разряды из упакованных значений с плавающей точкой двойной точности операнда источник (ХММ-регистр) и сохраняет полученную знаковую маску в двух младших битах операнда приемник (32-разрядный общий регистр).

MOVMSKPS

- MOVMSKPS приемник, источник
- 0F,50,/r MOVMSKPS r32, rxmm
- Перемещение знаковой маски в целочисленный регистр.

Действие: команда формирует маску из знаковых разрядов четырех чисел с плавающей точкой в коротком формате, упакованных в ХММ-регистр (источник). После операции содержимое всего 32-разрядного регистра (приемника) изменяется следующим образом: в его младшую тетраду заносится знаковая маска, остальные разряды регистра обнуляются.



MOVNTDQ

- MOVNTDQ приемник, источник
- 66 0F E7 /r MOVNTDQ m128, rxmm
- Сохранение двойного учетверенного слова из ХММ-регистра в памяти без использования кэша.

MOVNTI

- MOVNTI приемник, источник

- 0F C3 /r MOVNTI m32, r32
- Сохранение двойного слова из 32-разрядного регистра общего назначения в памяти без использования кэша.

MOVNTPD

- MOVNTPD приемник, источник
- 66 0F 2B /r MOVNTPD m128, rxmm
- Сохранение упакованных значений с плавающей точкой двойной точности из XMM-регистра в памяти без использования кэша.

MOVQ

- MOVQ приемник, источник
- См. описание команды MOVQ в разделе «Команды блока MMX».

MOVQ2DQ

- MOVQ2DQ приемник, источник
- F3 0F D6 MOVQ2DQ rxmm, rmmx
- Перемещение учетверенного слова из MMX-регистра в младшее учетверенное слово XMM-регистра.

Действие: команда перемещает содержимое источника в разряды 0...63 приемника, а в разряды 64... 127 приемника заносит значение 0000000000000000h.

MOVNTPS

- MOVNTPS приемник, источник
- 0F 2B /r MOVNTPS m128,rxmm
- Запись в намять 128 бит из XMM-регистра, минуя кэш.

MOVSD

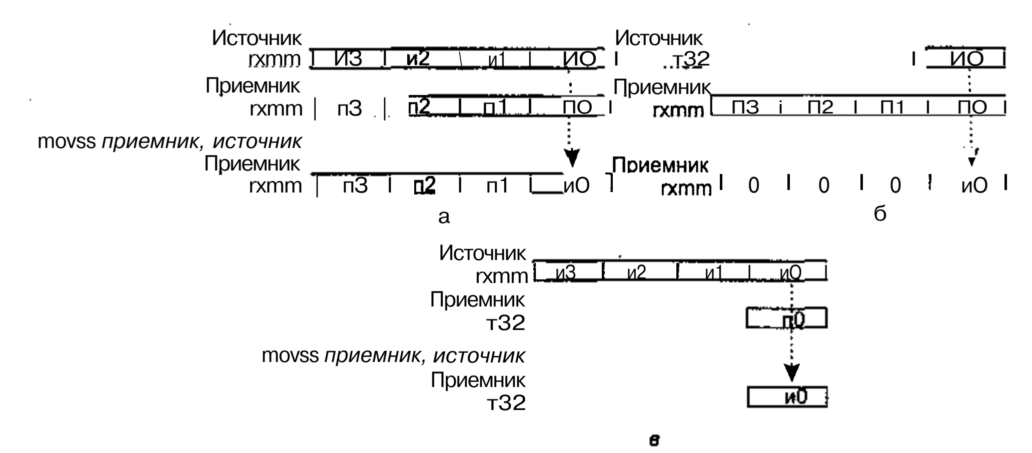
- MOVSD приемник, источник
- F2 0F 10/r MOVSD rxmm1, rxmm2/m64
  F2 0F 11 /r MOVSD rxmm2/m64, rxmm1
- Перемещение скалярного значения с плавающей точкой двойной точности.

Действие: команда перемещает скалярное значение с плавающей точкой двойной точности из разрядов 0...63 источника в разряды 0...63 приемника. Если операнды — XMM-регистры, то разряды 64...127 приемника не меняются. Если источник — ячейка памяти, то разряды 64...127 приемника обнуляются.

MOVSS

- MOVSS приемник, источник
- F3 0F 10 /r MOVSS rxmm1, rxmm2/m32
  F3 0F 11 /r MOVSS rxmm2/m32, rxmm1
- Перемещение скалярных значений в формате XMM.

Действие: команда копирует младшие 32 бита источника в младшие 32 бита приемника. Если используется операнд в памяти, то в команде указывается адрес, соответствующий адресу младшего байта данных в памяти. Если операнд в памяти используется в качестве источника, то эти 32 бита копируются в младшее двойное слово 128-разрядного приемника — XMM-регистра, остальные 96 бит этого регистра устанавливаются в 0.



MOVUPD

- MOVUPD приемник, источник
- 66 OF 10 /r MOVUPD xmm1, xmm2/m128
  66 OF 11 /r MOVUPD xmm2/m128, xmm1
- Перемещение невыровненных упакованных значений с плавающей точкой двойной точности.

Действие: команда перемещает два двойных учетверенных слова (содержащих два упакованных значения с плавающей точкой двойной точности) из источника в приемник. Выравнивания операнда в памяти по 16-разрядной границе не требуется.

MOVUPS

- MOVUPS приемник, источник
- OF,10,/r MOVUPS xmm1, xmm2/ml28
  OF,ll,/r MOVUPS xmm2/m128, xmml
- Перемещение невыровненных упакованныхзначений в формате XMM.

Действие: команда перемещает 128 бит источника в соответствующие биты приемника.

MULPD

- MULPD приемник, источник
- 66 OF 59 /r MULPD rxmm1, rxmm2/m128
- Умножение упакованных значений с плавающей точкой двойной точности.

Действие: команда умножает пары упакованных значений с плавающей точкой двойной точности источника и приемника по следующей схеме: разряды 0...63 приемника и источника перемножаются и помещаются в разряды 0...63 приемника, разряды 64...127 приемника и источника перемножаются и помещаются в разряды 64...127 приемника.

MULPS

- MULPS приемник, источник
- OF,59,/r MULPS xmm1, xmm2/ml28
- Умножение упакованных значений в формате XMM.

Действие: команда умножает четыре пары вещественных чисел в коротком формате. Схема работы команды MULPS показана на следующем рисунке.

mulps приемник, источник

Приемник rxmm	п3	п2	п1	п0
	⊗	⊗	⊗	⊗
Источник rxmm/m128	и3	и2	и1	и0
	=	=	=	=
Приемник rxmm	п3*и3	п2*и2	п1*и1	п0*и0

MULSD

- MULSD приемник, источник
- F2 0F 59 /r MULSD rxmm1, rxmm2/m64
- Умножение скалярных упакованных значений с плавающей точкой двойной точности.

Действие: команда умножает младшие упакованные значения с плавающей точкой двойной точности источника и приемника по следующей схеме: разряды 0...63 приемника и источника перемножаются и помещаются в разряды 0...63 приемника, разряды 64...127 приемника не меняются.

MULSS

- MULSS приемник, источник
- F3 0F 59 /r MULSS rxmm1, rxmm2/m32
- Умножение скалярных значений в формате XMM.

Действие: операнды источник и приемник находятся в XMM-регистре, кроме того, операнд источник может находиться в 32-разрядной ячейке памяти. Команда умножает вещественные значения в младших парах операндов в формате XMM по следующей схеме: разряды 0...31 приемника и источника перемножаются и помещаются в разряды 0...31 приемника, разряды 32...127 не меняются.

ORPD

- ORPD приемник, источник
- 66 0F 56 /r ORPD xmm1, xmm2/m128
- Поразрядное логическое ИЛИ над упакованными значениями с плавающей точкой двойной точности.

Действие: команда выполняет операцию поразрядного логического ИЛИ над парами упакованных значений с плавающей точкой двойной точности в разрядах 0...127 приемника и источника и помещает результат в разряды 0...127 приемника.

ORPS

- ORPS приемник, источник
- 0F 56 /r ORPS rxmm1,rxmm2/m128
- Поразрядное логическое ИЛИ над каждой парой битов упакованных операндов источник и приемник в формате XMM.

PACKSSWB/PACKSSDW

- PACKSSWB приемник, источник
  PACKSSDW приемник, источник
- См. описание команд PACKSSWB/PACKSSDW в разделе «Команды блока MMX».

PACKUSWB

- PACKUSWB приемник, источник
- См. описание команды PACKUSWB в разделе «Команды блока MMX».

PADDB/PADDW/PADDD

- PADDB приемник, источник
  PADDW приемник, источник
  PADDD приемник, источник
- См. описание команд PADDB/PADDW/PADDD в разделе «Команды блока MMX».

PADDQ

- PADDQ приемник, источник
- См. описание команды PADDQ в разделе «Команды блока MMX».

PADDSB/PADDSW

- PADDSB приемник, источник
  PADDSW приемник, источник
- См. описание команд PADDSB/PADDSW в разделе «Команды блока MMX».

PADDUSB/PADDUSW

- PADDUSB приемник, источник
  PADDUSW приемник, источник
- См. описание команд PADDUSB/PADDUSW в разделе «Команды блока MMX».

PAND

- PAND приемник, источник
- См. описание команды PAND в разделе «Команды блока MMX».

PANDN

- PANDN приемник, источник
- См. описание команды PANDN в разделе «Команды блока MMX».

PAVGB/PAVGW

- PAVGB приемник, источник
  PAVGW приемник, источник
- См. описание команд PAVGB/PAVGW в разделе «Команды блока MMX».

PCMPEQB/PCMPEQW/PCMPEQD

- PCMPEQB приемник, источник
  PCMPEQW приемник, источник
  PCMPEQD приемник, источник
- См. описание команд PCMPEQB/PCMPEQW/PCMPEQD в разделе «Команды блока MMX».

PCMPGTB/PCMPGTW/PCMPGTD

- PCMPGTB приемник, источник
  PCMPGTW приемник, источник
  PCMPGTD приемник, источник
- См. описание команд PCMPGTB/PCMPGTW/PCMPGTD в разделе «Команды блока MMX».

PEXTRW

- PEXTRW приемник, источник, маска
- См. описание команды PEXTRW в разделе «Команды блока MMX».

PINSRW

- PINSRW приемник, источник, маска
- См. описание команды PINSRW в разделе «Команды блока MMX».

PMADDWD

- PMADDWD приемник, источник
- См. описание команды PMADDWD в разделе «Команды блока MMX».

PMAXSW

- PMAXSW приемник, источник
- См. описание команды PMAXSW в разделе «Команды блока MMX».

PMAXUB

- PMAXUB приемник, источник
- См. описание команды PMAXUB в разделе «Команды блока MMX».

PMINSW

- PMINSW приемник, источник
- См. описание команды PMINSW в разделе «Команды блока MMX».

PMINUB

- PMINUB приемник, источник
- См. описание команды PMINUB в разделе «Команды блока MMX».

PMOVMSKB

- PMOVMSKB приемник, источник
- См. описание команды PMOVMSKB в разделе «Команды блока MMX».

PMULHUW

- PMULHUW приемник, источник
- См. описание команды PMULHUW в разделе «Команды блока MMX».

PMULHW

- PMULHW приемник, источник
- См. описание команды PMULHW в разделе «Команды блока MMX».

PMULLW

- PMULLW приемник, источник
- См. описание команды PMULLW в разделе «Команды блока MMX».

PMULUDQ

- PMULUDQ приемник, источник
- См. описание команды PMULUDQ в разделе «Команды блока MMX».

POR

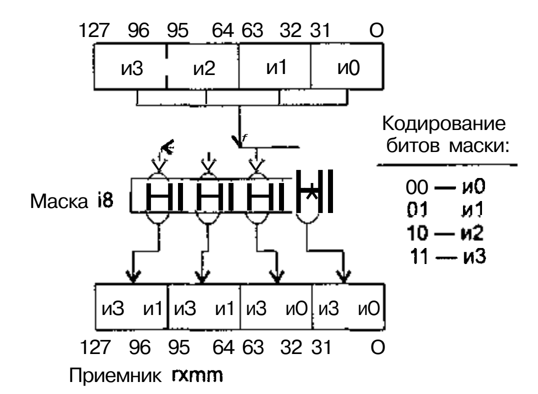
- POR приемник, источник
- См. описание команды POR в разделе «Команды блока MMX».

PSADBW

- PSADBW приемник, источник
- См. описание команды PSADBW в разделе «Команды блока MMX».

PSHUFD

- PSHUFD приемник, источник, маска
- *66 0F 70 /r* i8 PSHUFD *rxmm1,rxmm2/m128,i8*
- Копирование двойных слов из операнда источник (XMM-регистр) в операнд приемник (XMM-регистр).



Действие: на основе значения пар битов маски команда копирует двойные слова из источника в приемник. Каждая пара битов маски определяет номер слова источника для перемещения в приемник следующим образом:

- Маска[1:0]:
  - 00 - приемник[0...31] ← источник[0...31];
  - 01 - приемник[0...31] ← источник[32...63];
  - 10 — приемник[0...31] ← источник[64...95];
  - 11 - приемник[0...31] ← источник[96...127].
- Маска[3:2]:
  - 00 - приемник[32...63] ← источник[0...31];
  - 01 - приемник[32...63] ← источник[32...63];
  - 10 - приемник[32...63] ← источник[64...95];
  - 11 - приемник[32...63] ← источник[96...127].
- Маска[5:4]:
  - 00 — приемник[64...95] ← источник[0...31];
  - 01 - приемник[64...95] ← источник[32...63];
  - 10 - приемник[64...95] ← источник[64...95];
  - 11 - приемник[64...95] ← источник[96...127].
- Маска[7:6]:
  - 00 - приемник[96...127] ← источник[0...31];

D 01 - приемник[96...127] ← источник[32...63];

□ 10 - приемник[96...127] ← источник[64...95];

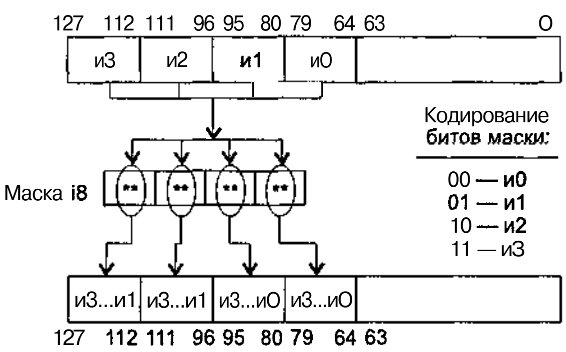
D 11 - приемник[96...127] ← источник[96...127].

Если использовать один и тот же XMM-регистр в качестве источника и приемника, то можно выполнять любые перестановки двойных слов в пределах одного XMM-регистра, в том числе и инициализацию значением одного двойного слова других двойных слов. Работу команды PSHUFD поясняет схема (см. предыдущую страницу).

PSHUFHW

- PSHUFHW приемник, источник, маска
- F3 0F 70 /r i8 PSHUFHW rxmm1,rxmm2/m128,i8
- Копирование слов из старшего учетверенного упакованного слова операнда источник (XMM-регистр) в старшее учетверенное упакованное слово операнда приемник (XMM-регистр).

Действие: на основе значения пар битов маски команда копирует слова из старшего упакованного учетверенного слова источника в старшее учетверенное упакованное слово приемника. Каждая пара битов маски определяет номер слова источника для перемещения в приемник следующим образом:



- Маска[1:0]:
  - 00 — приемник[64...79] ← источник[64...79];
  - 01 — приемник[64...79] ← источник[80...95];
  - 10 — приемник[64...79] ← источник[96...111];
  - 11 — приемник[64...79] ← источник[112...127].
- Маска[3:2]:
  - 00 — приемник[80...95] ← источник[64...79];
  - 01 — приемник[80...95] ← источник[80...95];
  - 10 — приемник[80...95] ← источник[96...111];
  - 11 — приемник[80...95] ← источник[112...127].
- Маска[5:4):
  - 00 — приемник[96...111] ← источник[64...79];
  - 01 — приемник[96...111] ← источник[80...95];
  - 10 — приемник[96...111] ← источник[96...111];

D И - приемник[96...111] ← источник[112...127].

№ Маска[7:6]:

□ 00 - приемник[112...127] ← источник[64...79];

□ 01 - приемник[112...127] ← источник[80...95];

D 10- приемник[112...127] ← источник[96...111];

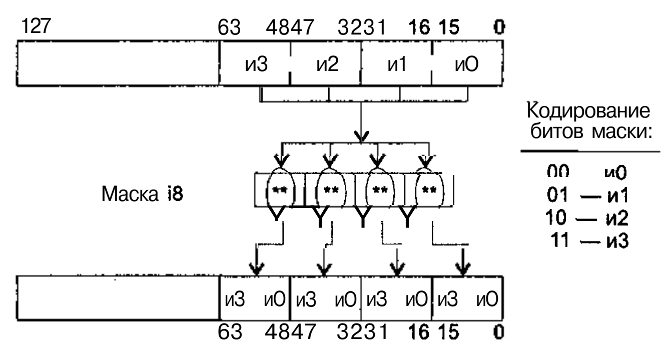
П И - приемник[112...127] ← источник[112...127].

Если использовать один и тот же XMM-регистр в качестве источника и приемника, то можно выполнять любые перестановки слов в пределах старшего учетверенного слова одного XMM-регистра, в том числе и инициализацию значением одного слова других слов. Работу команды PSHUFHW поясняет схема (см. предыдущую страницу).

PSHUFLW

- PSHUFLW приемник, источник, маска
- F2 0F 70 /r i8 PSHUFLW rxmml,rxmm2/ml28,i8
- Копирование слов из младшего учетверенного упакованного слова операнда источник (XMM-регистр) в младшее учетверенное упакованное слово операнда приемник (XMM-регистр).

Действие: на основе значения пар битов маски команда копирует слова из младшего учетверенного слова источника в младшее учетверенное слово приемника. Каждая пара битов маски определяет номер слова источника для перемещения в приемник следующим образом:



Маска[1:0]:

П 00 - приемник[00...15] ← источник[00...15];

П 01 — приемник[00...15] ← источник[16...31];

О 10 — приемник[00...15] ← источник[32...47];

□ 11 — приемник[00...15] ← источник[48...63].

Маска[3:2]:

D 00 — приемник[16...31] ← источник[00...15];

П 01 — приемник[16...31] ← источник[16...31];

П 10 — приемник[16...31] ← источник[32...47];

П 11 — приемник[16...31] ← источник[48...63].

Маска[5:4]:

□ 00 — приемник[32...47] ← источник[00...15];

П 01 — приемник[32...47] ← источник[16...31];

D 10 — приемник[32...47] ← источник[32...47];

- 11 - приемник[32...47] ← источник[48...63].
- Маска[7:6]:
  - 00 - приемник[47...63] ← источник[00...15];
  - 01 - приемник[47...63] ← источник[16...31];
  - 10 - приемник[47...63] ← источник[32...47];
  - 11 — приемник[47...63] ← источник[48...63].

Если использовать один и тот же XMM-регистр в качестве источника и приемника, то можно выполнять любые перестановки слов в пределах младшего учетверенного слова одного XMM-регистра, в том числе и инициализацию значением одного слова других слов. Работу команды PSHUFLW поясняет следующая схема (см. предыдущую страницу).

PSLLDQ

- PSLLDQ приемник, количество_сдвигов
- 66 0F 73 /7 i8 PSLLDQ rxmml,imm8
- Логический сдвиг влево приемника на заданное число байтов.

Действие: сдвиг влево приемника на число байтов, указанных непосредственным операндом количество_сдвигов. Освобождаемые слева младшие байты обнуляются. Если значение, указанное операндом количество_сдвигов, больше, чем 15, операнд приемник обнуляется.

PSLLW/PSLLD/PSLLQ

- PSLLW приемник, количество_сдвигов
  PSLLD приемник, количество_сдвигов
  PSLLQ приемник, количество_сдвигов
- См. описание команд PSLLW/PSLLD/PSLLQ в разделе «Команды блокаMMX».

PSRAW/PSRAD

- PSRAW приемник, количество_сдвигов
  PSRAD приемник, количество_сдвигов
- См. описание команд PSRAW/PSRAD в разделе «Команды блока MMX».

PSRLDQ

- PSRLDQ приемник, количество_сдвигов
- 66 0F 73 /3 i8 PSRLDQ rxmml,i8
- Сдвиг вправо приемника на заданное число байтов.

Действие: сдвиг вправо приемника на число байтов, указанных непосредственным операндом количество_сдвигов. Освобождаемые справа младшие байты обнуляются. Если значение, указанное операндом количество_сдвигов, больше, чем 15, операнд приемник обнуляется.

PSRLW/PSRLD/PSRLQ

- PSRLW приемник, количество_сдвигов
  PSRLD приемник, количество_сдвигов
  PSRLQ приемник, количество_сдвигов
- См. описание команд PSRLW/PSRLD/PSRLQ в разделе «Команды блока MMX».

PSUBB/PSUBW/PSUB

- PSUBB приемник, источник
  PSUBW приемник, источник
  PSUBD приемник, источник
- См. описание команд PSUBB/PSUBW/PSUBD в разделе «Команды блока MMX».

PSUBQ

- PSUBQ приемник, источник
- См. описание команды PSUBQ в разделе «Команды блока MMX».

PSUBSB/PSUBSW

- PSUBSB приемник, источник
  PSUBSW приемник, источник
- См. описание команд PSUBSB/PSUBSW в разделе «Команды блока MMX».

PSUBUSB/PSUBUSW

- PSUBUSB приемник, источник
  PSUBUSW приемник, источник
- См. описание команд PSUBUSB/PSUBUSW в разделе «Команды блока MMX».

PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ

- PUNPCKHBW приемник, источник
  PUNPCKHWD приемник, источник
  PUNPCKHDQ приемник, источник
  PUNPCKHQDQ приемник, источник
- См. описание команд PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ в разделе «Команды блока MMX».

PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ

- PUNPCKLBW приемник, источник
  PUNPCKLWD приемник, источник
  PUNPCKLDQ приемник, источник
  PUNPCKLQDQ приемник, источник
- См. описание команд PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ в разделе «Команды блока MMX».

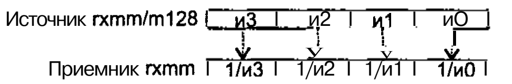
PXOR

- PXOR приемник, источник
- см. описание команды PXOR в разделе «Команды блока MMX».

RCPPS

- RCPPS приемник, источник
- 0F 53 /r RCPPS rxmm1,rxmm2/m128
- Вычисление обратных упакованных значений в формате XMM.

Действие: команда вычисляет обратные значения элементов источника по формуле 1/(элемент источника). Максимальная ошибка вычисления не должна превышать значения $1{,}5 \cdot 2^{-12}$. Схема работы команды RCPPS показана на следующем рисунке.



RCPSS

- RCPSS приемник, источник
- F3 0F 53 /r RCPSS rxmml,rxmm2/m32
- Скалярное вычисление обратного упакованного значения в формате XMM.

Действие: команда вычисляет обратное значение младшего элемента операнда источник по формуле 1/(элемент источника). Максимальная ошибка вычисления не должна превышать значения $1{,}5 \cdot 2^{-12}$.

RSQRTPS

- RSQRTPS приемник, источник
- 0F 52 /r RSQRTPS rxmm1,rxmm2/m128
- Вычисление обратного значения квадратного корня от упакованного значения в формате XMM.

Действие: команда вычисляет сначала квадратный корень каждого элемента операнда источник, а затем — обратную величину полученного значения:

$$1/\sqrt{\text{элемент источника}}.$$

Максимальная ошибка вычисления не должна превышать значения $1{,}5 \cdot 2^{-12}$.

RSQRTSS

- RSQRTSS приемник, источник
- F3 0F 52 /r RSQRTSS rxmm1,rxmm2/m32
- Скалярная аппроксимация обратных значений квадратного корня упакованных значений в формате XMM.

Действие: команда для младшего элемента операнда источник сначала вычисляет квадратный корень, а затем — обратную величину полученного значения:

$$1/\sqrt{\text{элемент источника}}.$$

Максимальная ошибка вычисления не должна превышать значения $1{,}5 \cdot 2^{-12}$. Старшие элементы операнда приемник не меняются.

SHUFPD

- SHUFPD приемник, источник, маска
- 66 0F C6 /r i8 SHUFPD rxmm1,rxmm2/m128,i8
- Перестановка упакованных значений с плавающей точкой двойной точности.

Действие: команда перемещает упакованные значения с плавающей точкой двойной точности из приемника и источника в приемник в соответствии со значением непосредственного операнда маска. Биты маски определяют номера упакованных значений с плавающей точкой двойной точности в источнике или приемнике, которые будут перемещены в приемник следующим образом:

- Маска[0]:
  - 0 — приемник[0...63] ← приемник[0...63];
  - 1 — приемник[0...63] ← приемник[64...127].
- Маска[1]:
  - 0 — приемник[64...127] ← источник[0...63];
  - 1 — приемник[64...127] ← источник[64...127].

Для перестановки в пределах одного регистра можно использовать один и тот же XMM-регистрвкачествеисточникаиприемника.

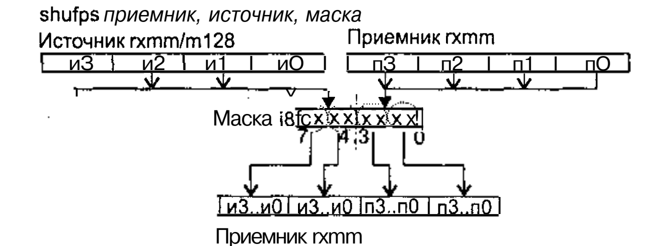
SHUFPS

- SHUFPS приемник, источник, маска
- 0F C6 /r ib SHUFPS rxmm1,rxmm2/m128,imm8
- Перераспределение упакованных значений в формате XMM.

Действие: команда перераспределяет любые два из четырех двойных слов приемника в два младших двойных слова того же приемника и любые два из четырех двойных слов источника в два старших двойных слова приемника. Если использовать один и тот же XMM-регистр в качестве источника и приемника, то можно выполнять любые перестановки в пределах одного регистра. Каждая пара битов маски определяет номер двойного слова источника или приемника, которое будет перемещено в приемник следующим образом:

- Маска[1:0]:
  - 00 приемник[00...31] ← источник[00...31];
  - 01 приемник[00...31] ← источник[32...63];
  - 10 приемник[00...31] ← источник[64...95];
  - 11 приемник[00...31] ← источник[96...127].
- Маска[3:2]:
  - 00 приемник[32...63] ← источник[00...31];
  - 01 приемник[32...63] ← источник[32...63];
  - 10 приемник[32...63] ← источник[64...95];
  - 11 приемник[32...63] ← источник[96...127].
- Маска[5:4]:
  - 00 приемник[64...95] ← источник[00...31];
  - 01 приемник[64...95] ← источник[32...63];
  - 10 приемник[64...95] ← источник[64...95];
  - 11 приемник[64...95] ← источник[96...127].
- Маска[7:6]:
  - 00 приемник[96...127] ← источник[00...31];
  - 01 приемник[96...127] ← источник[32...63];
  - 10 приемник[96...127] ← источник[64...95];
  - 11 приемник[96...127] ← источник[96...127].

Схематически работа команды SHUFPS показана на следующем рисунке.



SQRTPD

- SQRTPD приемник, источник
- 66 0F 51 /r SQRTPD rxmm1,rxmm2/m128

- Вычисление квадратного корня упакованных значений с плавающей точкой двойной точности.

Действие: команда вычисляет квадратный корень упакованных значений с плавающей точкой двойной точности источника по следующей схеме: вычисляется значение квадратного корня в разрядах 0...63 источника и помещается в разряды 0...63 приемника, вычисляется значение квадратного корня в разрядах 64...127 источника и помещается в разряды 64...127 приемника.

SQRTPS

- SQRTPS приемник, источник
- 0F 51 /r SQRTPS rxmm1,rxmm2/m128
- Вычисление квадратного корня упакованных значений в формате XMM.

Действие: команда извлекает квадратный корень из четырех упакованных вещественных чисел в коротком формате.

SQRTSD

- SQRTSD приемник, источник
- F2 0F 51 /r SQRTSD rxmm1,rxmm2/m64
- Вычисление квадратного корня скалярного упакованного значения сплавающей точкой двойной точности.

Действие: команда вычисляет значение квадратного корня младшего упакованного значения с плавающей точкой двойной точности источника по следующей схеме: вычисляется значение квадратного корня в разрядах 0...63 источника и помещается в разряды 0...63 приемника, значение в разрядах 64...127 приемника не меняется.

SQRTSS

- SQRTSS приемник, источник
- F3 0F 51 /r SQRTSS rxmml,rxmm2/m32
- Скалярное извлечение квадратного корня.

Действие: команда извлекает квадратный корень из младшего двойного слова операнда источник, который должен представлять собой упакованное вещественное число в формате XMM.

STMXCSR

- STMXCSR приемник
- 0F AE /3 STMXCSR m32
- Сохранение регистра управления/состояния MXCSR в 32-разрядной ячейке памяти.

SUBPD

- SUBPD приемник, источник
- 66 0F 5C/r SUBPD rxmm1,rxmm2/m128
- Вычитание упакованных значений с плавающей точкой двойной точности.

Действие: команда вычитает пары упакованных значений с плавающей точкой двойной точности источника и приемника по следующей схеме: из разрядов 0...63 приемника вычитаются разряды 0...63 источника и результат помещается в разряды 0...63 приемника, из разрядов 64...127 приемника вычитаются разряды 64...127 источника и результат помещается в разряды 64... 127 приемника.

SUBPS

- SUBPS приемник, источник
- 0F 5C /r SUBPS rxmm1,rxmm2/m128

- Вычитание упакованных значений в формате XMM.

SUBSD

- SUBSD приемник, источник
- F2 0F 5C /r SUBSD rxmm1,rxmm2/m64
- Вычитание скалярных упакованных значений с плавающей точкой двойной точности.

Действие: команда вычитает младшие упакованные значения с плавающей точкой двойной точности источника и приемника по следующей схеме: из разрядов 0...63 приемника вычитаются разряды 0...63 источника и результат помещается в разряды 0...63 приемника, разряды 64...127 приемника не меняются.

SUBSS

- SUBSS приемник, источник
- F3 0F 5C /r SUBSS rxmm1,rxmm2/m32
- Вычитание скалярных значений в формате XMM.

Действие: команда вычитает значения младшей пары вещественных чисел в формате XMM.

UCOMISD

- UCOMISD приемник, источник, условие
- 66 0F 2E /r UCOMISD rxmm1,rxmm2/m64
- Сравнение неупорядоченных скалярных значений с плавающей точкой двойной точности и установка регистра EFLAGS.

Действие: команда сравнивает неупорядоченные скалярные значения с плавающей точкой двойной точности в разрядах 0...63 приемника и источника. По результату сравнения устанавливаются флаги ZF, PF, и CF в регистре EFLAGS (см. описание команды COMISD). Отличие команды COMISD от команды UCOMISD состоит в условиях генерации исключения недействительной операции с плавающей точкой (#I): команда COMISD генерирует его, когда приемник и/или источник являются нечислом (QNaN или SNaN); команда UCOMISD генерирует его только в том случае, когда один из исходных операндов является нечислом SNaN. В случае генерации немаскированного исключения недействительной операции с плавающей точкой регистр EFLAGS не модифицируется.

UCOMISS

- UCOMISS приемник, источник
- 0F 2E /r UCOMISS rxmm1,rxmm2/m32
- Неупорядоченное скалярное сравнение значений в формате XMM с установкой флагов в EFLAGS.

Таблица П.25. Допустимые значения флагов в регистре EFLAGS

Соотношение операндов	Значения флагов
Приемник > источник	OF-SF-AF-0; ZF-0; PF=0; CF-0
Приемник < источник	OF-SF-AF-0; ZF-0; PF-0; CF-1
Приемник = источник	OF-SF-AF-0; ZF=1; PF-0; CF=0
Приемник или источник — нечисло sNaN	OF-SF-AF-0; ZF=1; PF=1; CF=1

Действие: сравнение пары вещественных элементов в коротком формате, расположенных в младшем двойном слове операндов в формате XMM. В результате выполнения команды формируются значения флагов ZF, PF и CF, а флаги OF, SF и AF устанавливаются в 0 (табл. П.25). В про-

цессе работы команда распознает специальные значения qNaN и sNaN. При возникновении незамаскированных исключений расширения XMM регистр EFLAGS не изменяется.

UNPCKHPD

- UNPCKHPD приемник, источник
- 66 0F 15 /r UNPCKHPD rxmm1,rxmm2/m128.
- Разделение и чередование старших упакованных значений с плавающей точкой двойной точности.

Действие: команда разделяет старшие упакованные значения с плавающей точкой двойной точности в источнике и приемнике, помещая их с чередованием в приемник по следующей схеме: разряды 64...127 приемника помещаются в разряды 0...63 приемника, разряды 64...127 источника помещаются в разряды 64...127 приемника.

UNPCKHPS

- UNPCKHPS приемник, источник
- 0F 15 /r UNPCKHPS rxmm1,rxmm2/m128
- Чередование верхних упакованных значений в формате XMM.

Действие: перемещение путем чередования двух старших двойных слова приемника и источника. Младшие двойные слова приемника и источника игнорируются.

UNPCKLPD

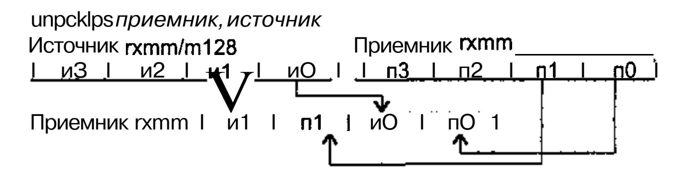
- UNPCKLPD приемник, источник
- 66 0F 14 /r UNPCKLPD rxmm1,rxmm2/m128
- Разделение и чередование младших упакованных значений с плавающей точкой двойной точности.

Действие: команда разделяет младшие упакованные значения с плавающей точкой двойной точности в источнике и приемнике, помещая их с чередованием в приемник по следующей схеме: разряды 0...63 приемника не меняются, разряды 0...63 источника помещаются в разряды 64...127 приемника.

UNPCKLPS

- UNPCKLPS приемник, источник
- 0F 14 /r UNPCKLPS rxmm1,rxmm2/m128
- Чередование нижних упакованных значений в формате XMM.

Действие: перемещение путем чередования двух младших двойных слова приемника и источника. Старшие двойные слова приемника и источника игнорируются.



XORPD

- XORPD приемник, источник
- 66 0F 57 /r XORPD rxmm1,rxmm2/m128
- Выполнение поразрядной операции логического исключающего ИЛИ над упакованными значениями с плавающей точкой двойной точности.

Действие: команда выполняет поразрядную операцию логического исключающего ИЛИ над парами упакованных значений с плавающей точкой двойной точности в разрядах 0...127 приемника и источника и результат помещает в разряды 0...127 приемника.

XORPS

- XORPS приемник, источник
- 0F 57 /r XORPS rxmm1,rxmm2/m128
- Выполнение поразрядной операции логического исключающего ИЛИ над упакованными значениями в формате XMM.

Список литературы

1. *Хаммел Р. Л.* Последовательная передача данных: Руководство для программиста. - М.: Мир, 1996.
2. *Тук М.* Аппаратные средства IBM PC: Энциклопедия. — СПб.: Питер, 2001.
3. *Тук М.* Интерфейсы ПК: Энциклопедия. — СПб.: Питер, 2001.
4. *Рихтер Дж.* Программирование серверных приложений для Windows 2000. — СПб.: Питер, 2001.
5. *Соломон Д.* Внутреннее устройство Windows 2000. — СПб.: Питер, 2001.
6. *Вильямс А.* Системное программирование Windows 2000. — СПб.: Питер, 2001.
7. *Таненбаум Э.* Архитектура компьютера. — СПб.: Питер, 2002.
8. *Юров В.* Assembler: Практикум. — СПб.: Питер, 2001.
9. *Касперский Крис.* Образ мышления — дизассемблер IDA. — М.: Солон-Р, 2001.
10. *Рихтер Дж.* Windows для профессионалов. — СПб.: Питер, 2000.

И. IA-32 Intel® Architecture Software Developer's Manual. Vol. 2. Instruction Set Reference. Intel Corporation, 2002.

12. Assembly-Language Developer System, Version 6.1, for MS-DOS and Windows Operation System Microsoft Corporation.
13. *Тук М., Юров В.* Процессоры Pentium IV, Athlon и Duron. — СПб.: Питер, 2001.
14. IA-32 Intel® Architecture Software Developer's Manual. Vol. 1. Basic architecture. Intel Corporation, 2002.
15. IA-32 Intel® Architecture Software Developer's Manual. Vol. 3. System programming guide. Intel Corporation, 2002.
16. *Кулаков В.* Программирование на аппаратном уровне: Спец. справ.. — СПб.: Питер, 2003.
17. Персональный компьютер в играх и задачах. — М.: Наука, 1988.
18. *Шилдт Г.* Теория и практика С++. - СПб.: БХВ-Петербург, 2001.

Алфавитный указатель

C

CISC-процессор, 31
CMOS-память, 156

M

make-файл, 403

R

RISC-технология разработки процессоров, 23

U

u-конвейер, 32

V

v-конвейер, 32

W

Windows-приложение
  каркасное, 367, 379
  консольное, 367
  оконное, 367

А

абсолютное выражение, 99
адресное пространство ввода-вывода, 149
адресный операнд, 92
аргументы, 330
  пролога/эпилога, 343
  фактические, 300, 330
  формальные, 300, 330
арифметические команды, 474
арифметический сдвиг, 200
архитектура
  компьютера фон Неймана, 28
  процессоров IA-32, 22, 29
  ЭВМ, 27
асинхронные сообщения Windows, 393, 399
атрибут комбинирования сегментов, 333

Б

база
  кадра стека, 161
  сегмента, 52
базовая индексная адресация со смещением, 272
байт, 49, 110
  масштаба, индекса и базы sib, 70
  режима адресации mod r/m, 64
безусловный переход, 214
бесконечность, 464
библиотека DLL, 220, 363
бит, 49, 194
битовое поле, 112
битовые строки, 205
блок
  декодирования команд, 22
  микропрограммного управления, 18, 19, 60
  предварительной выборки, 22
  предсказания переходов, 32
  сегментации, 22
  страничной адресации, 22
  удаления и восстановления, 34, 38
буфер
  команд, 34
  меток перехода, 35
  переупорядочивания запросов к памяти, 35

В

венгерская запись, 386
вещественная неопределенность, 465
внешнее устройство, 18

возврат результата из процедуры, 340, 355
выражение ассемблера, 91, 97
вычисление степени произвольного
числа, 492

Г

главная функция Windows-
приложения, 368
глобальная дескрипторная таблица, 56

Д

данные простого типа, 113
двоично-десятичный код, 168
неупакованный формат, 168, 184
упакованный формат, 168, 190
двоичное дополнение, 84
двоичные целые числа, 168
двойное слово, 111
двухмерный массив, 275
декодер, 36
денормализованные вещественные
числа, 463
дескриптор сегмента, 54
дескрипторная таблица векторов
прерываний, 56
десятичные числа, 168
диалоговая процедура, 424
диалоговая функция, 420
динамический анализ потока данных, 32
динамическое исполнение, 32
директива
%CONDS, 321
%CTLS, 321
%INCL, 321
%LIST, 321
%MACS, 322
%NOCONDS, 321
%NOCTLS, 321
%NOINCL, 321
%NOLIST, 321
%NOMACS, 322
%NOSYMS, 321
%OUT, 319
%SYMS, 321
.ENDW, 244
.ERRB (ERRIFB), 317
.ERRDEF (ERRIFDEF), 317
.ERRDIF (ERRIFDIF), 317
.ERRE (ERRIFE), 318
.ERRIDN (ERRIFIDN), 318
.ERRNB (ERRIFNB), 317
.ERRNDEF (ERRIFNDEF), 317
.ERRNZ (ERRIF), 318
.LALL, 322
.LFCOND, 321
директива *(продолжение)*
.LIST, 321
.MODEL, 384
.SALL, 322
.SFCONDS, 321
.TFCOND, 322
.UNTILCXZ, 244
.XALL, 322
.XLIST, 321
_asm, 345
ARG, 223, 351, 357, 358, 360
ASSUME, 106, 214
CATSTR, 296
DB, 114
DD, 114
DF, 115
DISPLAY, 313, 319
DP, 115
DQ, 115
DT, 115
DW, 114
ENDP, 219, 327
ERR, 316
ERRIFDIFI, 318
ERRIFIDNI, 318
EVEN, 508
EXITM, 305, 308, 314
EXTRN, 329, 337, 385
GOTO, 305, 308
IF, 309
IFB, 313
IFDEF, 311
IFDIF, 314
IFDIFI, 314
IFE, 309
IFIDN, 314
IFIDNI, 314
IFNB, 313
IFNDEF, 311
INCLUDE, 384
INSTR, 296
INVOKE, 341
IRP, 305, 307
IRPC, 305, 307
LABEL, 211, 270
LOCAL, 223, 303, 343, 357
LOCALS, 234, 384
MODEL, 107, 353
OPTION EPILOGUE, 343
OPTION PROLOGUE, 343
ORG, 213
PROC, 219, 327, 346
PROTO, 341
PUBLIC, 212, 329, 337
REPT, 270, 305, 306
RETURNS, 223

директива *(продолжение)*
SEGMENT, 104, 337
SIZESTR, 297
SUBSTR, 296
USES, 223, 357
WHILE, 305, 306
директивы
ассемблера, 86, 87, 133
вывода
блоков условного ассемблирования, 321
макрорасширений, 322
текста включаемых файлов, 321
выделения подстроки в строке, 296
генерации пользовательской ошибки, 315
определения
вхождения одной строки в другую, 296
длины строки в текстовом макросе, 297
повторения, 305, 306
резервирования и инициализации данных, 113
сегментации, 337
слияния строк, 296
управления
листингом, 134, 321
процессом генерации макрорасширений, 305, 308
условной компиляции, 308
дополнительный код, 83
драйвер расширенной памяти EMS, 22

Ж

жизненный цикл программы на ассемблере, 122

З

запись, 286
значение REQ, 302

И

идентификатор, 89
имя макрокоманды, 297
индексная адресация со смещением, 272
инициализация массива, 270
интерфейс
API, 367
GUI, 367
исключение, 179, 500
исключительная ситуация, 453
исполнительное устройство, 18
исполнительный блок, 22
исчисление высказываний, 194

К

кадр стека, 359
каркасное Windows-приложение, 367, 379
кисть, 390
класс окна, 389
ключевые слова ассемблера, 89
код операции, 60, 63, 87
кодировка
ANSI, 386
UNICODE, 386
коды условия C3...C0, 472
команда
AAA, 185, 513
AAD, 189, 513
AAM, 188, 513
AAS, 186, 513
ADC, 514
adc, 171
ADD, 514
add, 171
ADDPD, 587
ADDPS, 587
ADDSD, 587
ADDSS, 587
AND, 195, 289, 514
ANDNPD, 588
ANDNPS, 588
ANDPD, 588
ANDPS, 588
ARPL, 515
BOUND, 515
BSF, 197, 515
BSR, 197, 516
BSWAP, 516
BT, 516
bt, 198
BTC, 516
btc, 199
BTR, 517
btr, 198
BTS, 517
bts, 198
CALL, 517
call, 222
CBW, 182, 518
CDQ, 182, 518
CLC, 518
CLD, 253, 518
CLFLUSH, 518, 588
CLI, 519
CLTS, 519
CMC, 519
CMOVcc, 519
CMP, 225, 520

команда *(продолжение)*
CMPPD, 589
CMPPS, 589
CMPS, 250, 255, 520
CMPSB, 250, 258, 520
CMPSD, 250, 258, 520, 589
CMPSS, 590
CMPSW, 250, 258, 520
CMPXCHG, 521
CMPXCHG8B, 521
COMISD, 591
COMISS, 591
CPUID, 521
CVTDQ2PD, 592
CVTDQ2PS, 592
CVTPD2DQ, 592
CVTPD2PI, 593
CVTPD2PS, 593
CVTPI2PD, 594
CVTPI2PS, 594
CVTPS2DQ, 594
CVTPS2PD, 595
CVTPS2PI, 595
CVTSD2SI, 596
CVTSD2SS, 597
CVTSI2SD, 597
CVTSI2SS, 597
CVTSS2SD, 598
CVTSS2SI, 598
CVTTPD2DQ, 599
CVTTPD2PI, 598
CVTTPS2DQ, 599
CVTTPS2PI, 600
CVTTSD2SI, 600
CVTTSS2SI, 601
CWD, 182, 518
CWDE, 182, 518
DAA, 191, 523
DAS, 192, 524
DEC, 524
dec, 174
DIV, 524
div, 179
DIVPD, 601
DIVPS, 601
DIVSD, 602
DIVSS, 602
EMMS, 569
ENTER, 356, 525
F2XM1, 491, 553
FABS, 479, 553
FADD, 476, 554
FADDP, 476, 554
FBLD, 469, 554
FBSTP, 470, 554

команда *(продолжение)*
FCHS, 479, 555
FCLEX, 496, 555
FCMOVcc, 555
FCOM, 472, 555
FCOMI, 556
FCOMIP, 556
FCOMP, 472, 555
FCOMPP, 472
FCOS, 483, 557
FDECSTP, 497, 557
FDIV, 477, 557
FDIVP, 478
FDIVR, 478, 558
FDIVRP, 478
FFREE, 558
FIADD, 474, 554
FICOM, 472, 558
FICOMP, 472, 558
FIDIV, 475
FIDIVR, 475
FILD, 469, 559
FIMUL, 475
FINCSTP, 497, 559
FINIT, 559
FIST, 469, 559
FISTP, 469, 559
FISUB, 474, 566
FISUBR, 475, 567
FLD, 469, 560
FLD1, 471, 560
FLDCW, 496, 560
FLDCWR, 481
FLDENV, 499, 561
FLDL2E, 471, 560
FLDL2T, 471, 560
FLDLG2, 471, 560
FLDLN2, 471, 560
FLDPI, 471, 560
FLDZ, 471, 560
FMUL, 477, 561
FMULP, 477
FNCLEX, 496, 555
FNINIT, 559
FNOP, 497, 561
FNSAVE, 497
FNSTCW, 496, 565, 566
FNSTENV, 498, 565
FNSTSW, 496
FPATAN, 483, 484, 562
FPREM, 490, 562
FPREM1, 490, 562
FPTAN, 483, 563
FRNDINT, 481, 563
FRSTOR, 497, 563

команда *(продолжение)*
FRSTPM, 500
FSAVE, 497, 564
FSCALE, 492, 564
FSETPM, 500
FSIN, 483, 564
FSINCOS, 483, 565
FSQRT, 479, 565
FST, 469, 565
FSTCW, 496, 565, 566
FSTCWR, 481
FSTENV, 498, 565
FSTP, 469, 565
FSTSW, 496
FSUB, 476, 566
FSUBP, 477, 566
FSUBR, 477, 567
FSUBRP, 477, 567
FTST, 472, 567
FUCOM, 472, 567
FUCOMI, 556
FUCOMIP, 556
FUCOMP, 472, 567
FUCOMPP, 472, 567
FWAIT, 453, 495, 568
FXAM, 464, 474, 568
FXCH, 569
FXRSTOR, 602
FXSAVE, 602
FXTRACT, 479, 569
FYL2X, 492, 569
FYL2XP1, 494, 569
GETFIELD, 290
HLT, 525
IDIV, 180,525
IMUL, 525
imul, 179
IN, 526
INC, 171,526
INS, 250, 265, 527
INSB, 250, 265, 527
INSD, 250, 265, 527
INSW, 250, 265, 527
INT 3, 527
INTO, 527
INVD, 528
INVLPG, 528
IRET, 528
IRETD, 528
Jcc, 528
jcc, 224
JCXZ, 229
JECXZ, 229
JMP, 215, 220, 241, 530
LAHF, 531

команда *(продолжение)*
LAR, 531
LDMXCSR, 603
LDS, 156, 252, 532
LEA, 156, 532
LEAVE, 356, 533
LES, 156, 252, 532
LFENCE, 603
LFS, 157, 532
LGDT, 533
LGS, 157, 532
LIDT, 533
LLDT, 533
LMSW, 533
LOCK, 533
LODS, 250, 261, 533
LODSB, 250, 262, 533
LODSD, 250, 262, 533
LODSW, 250, 262, 533
LOOP, 230, 534
LOOPE, 230
LOOPNE, 231
LOOPNZ, 231
LOOPZ, 230
LOOPcc, 534
LSL, 534
LSS, 157, 532
LTR, 534
MASKMOVDQU, 603
MASKMOVQ, 570
MAXPD, 604
MAXPS, 604
MAXSD, 604
MAXSS, 605
MFENCE, 605
MINPD, 605
MINPS, 605
MINSD, 605
MINSS, 606
MOV, 69, 147, 534, 535
MOVAPD, 606
MOVAPS, 606
MOVD, 570, 606
MOVDQ2Q, 606
MOVDQA, 606
MOVDQU, 606
MOVHLPS, 607
MOVHPD, 607
MOVHPS, 607
MOVLHPS, 607
MOVLPD, 607
MOVLPS, 608
MOVMSKPD, 608
MOVMSKPS, 608
MOVNTDQ, 608

команда *(продолжение)*
MOVNTI, 608
MOVNTPD, 609
MOVNTPS, 609
MOVNTQ, 570
MOVQ, 570, 609
MOVQ2DQ, 609
MOVS, 250, 253, 535
MOVSB, 250, 254, 535
MOVSD, 250, 254, 535, 609
MOVSS, 609
MOVSW, 250, 254, 535
MOVSX, 182, 536
MOVUPD, 610
MOVUPS, 610
MOVZX, 182, 536
MUL, 536
mul, 177
MULPD, 610
MULPS, 610
MULSD, 611
MULSS, 611
NEG, 175, 183, 536
NOP, 537
NOT, 196, 537
OR, 195, 289, 537
ORPD, 611
ORPS, 611
OUT, 537
OUTBS, 251
OUTDS, 251
OUTS, 250, 265, 538
OUTSB, 266, 538
OUTSD, 266, 538
OUTSW, 266, 538
OUTWS, 251
PACKSSDW, 571, 611
PACKSSWB, 571, 611
PACKUSWB, 572, 611
PADDB, 572, 612
PADDD, 572, 612
PADDQ, 573, 612
PADDSB, 573, 612
PADDSW, 573, 612
PADDUSB, 573, 612
PADDUSW, 573, 612
PADDW, 572, 612
PAND, 574, 612
PANDN, 574, 612
PAUSE, 538
PAVGB, 574, 612
PAVGW, 574, 612
PCMPEQB, 574, 612
PCMPEQD, 574, 612
PCMPEQW, 574, 612

команда *(продолжение)*
PCMPGTB, 575, 612
PCMPGTD, 575, 612
PCMPGTW, 575, 612
PEXTRW, 575, 612
PINSRW, 575, 613
PMADDWD, 575, 613
PMAXSW, 576, 613
PMAXUB, 576, 613
PMINSW, 576, 613
PMINUB, 576, 613
PMOVMSKB, 576, 613
PMULHUW, 577, 613
PMULHW, 578, 613
PMULLW, 578, 613
PMULUDQ, 579, 613
POP, 148, 162, 538
POPA, 163, 538
POPAD, 163, 538
POPAW, 163
POPF, 164, 539
POPFD, 164, 539
POPFW, 164
POR, 579, 613
PREFETCHNTn, 539
PSADBW, 580, 614
PSHUFD, 614
PSHUFHW, 615
PSHUFLW, 616
PSHUFW, 580
PSLLD, 581, 617
PSLLDQ, 617
PSLLQ, 581, 617
PSLLW, 581, 617
PSRAD, 581, 617
PSRAW, 581, 617
PSRLD, 581, 617
PSRLDQ, 617
PSRLQ, 581, 617
PSRLW, 581, 617
PSUBB, 582, 617
PSUBD, 582, 617
PSUBQ, 582, 618
PSUBSB, 583, 618
PSUBSW, 583, 618
PSUBUSB, 583, 618
PSUBUSW, 583, 618
PSUBW, 582, 617
PUNPCKHBW, 618
PUNPCKHDQ, 618
PUNPCKHQDQ, 618
PUNPCKHWD, 618
PUNPCKLBW, 585, 618
PUNPCKLDQ, 585, 618
PUNPCKLQDQ, 585, 618

команда *(продолжение)*
PUNPCKLWD, 585, 618
PUSH, 148, 162, 540
PUSHA, 162,540
PUSHAD, 163, 540
PUSHAW, 162
PUSHF, 164, 540
PUSHFD, 164, 540
PUSHFW, 164
PXOR, 587, 618
RCL, 203, 540
RCPPS, 618
RCPSS, 619
RCR, 203, 540
RDMSR, 541
RDPMC, 541
RDTSC, 541
REP, 251, 542
REPE, 251, 542
REPNE, 251, 542
REPNZ, 251, 542
REPZ, 251, 542
RET, 222, 542
ROL,201, 543
ROR, 201, 543
RSM, 543
RSQRTPS, 619
RSQRTSS, 619
SAHF, 544
SAL, 200, 544
SAR, 200, 544
SBB, 545
sbb, 174
SCAS, 250, 259, 545
SCASB, 250, 260, 545
SCASD, 250, 260, 545
SCASW, 250, 260, 545
SETcc, 229, 545
SETFIELD, 290
SFENCE, 545
SGDT, 546
SHL, 199, 546
SHLD, 204, 547
SHR, 200, 288, 546
SHRD, 547
shrd, 204
SHUFPD, 619
SHUFPS, 620
SLDT, 548
SMSW, 548
SQRTPD, 620
SQRTPS, 621
SQRTSD, 621
SQRTSS, 621
STC, 548

команда *(продолжение)*
STD, 253, 548
STI, 548
STMXCSR, 621
STOS, 250, 263, 548
STOSB, 250, 264, 548
STOSD, 250, 264, 548
STOSW, 250, 264, 548
STR, 549
SUB, 549
sub, 174
SUBPD, 621
SUBPS, 621
SUBSD, 622
SUBSS, 622
SYSENTER, 549
SYSEXIT, 550
TEST, 196,551
UCOMISD, 622
UCOMISS, 622
UD2, 551
UNPCKHPD, 623
UNPCKHPS, 623
UNPCKLPD, 623
UNPCKLPS, 623
VERR, 551
WAIT, 453, 495, 552
WBINVD, 552
WRMSR, 552
XADD, 183, 552
XCHG, 149, 552
XLAT, 158,553
XLATB, 553
XOR, 196, 553
XORPD, 624
XORPS, 624
командная строка
ml, 142
tasm, 129
tlink, 134
команды
арифметические, 474
вещественные, 476
дополнительные, 479
целочисленные, 474
ассемблера, 86, 133
загрузки констант, 471
обработки строк символов, 249
передачи данных, 468
сдвига, 199, 288
сдвига двойной точности, 203
сравнения данных, 472
трансцендентных функций, 482
управления сопроцессором, 494
цепочечные, 249

комментарий
в программе на ассемблере, 86
в теле макроопределения, 305
компилятор ресурсов, 403
компоновка программы, 134
конвейер, 31
конвейеризация вычислений, 22
конечный автомат, 245
консольное Windows-приложение, 367, 436
константа, 330
конструкция
.BREAK, 244
.CONTINUE, 244
.IF, 242
.REPEAT, 243
.WHILE, 244
контекст процедуры, 222
контроллер DMA, 265
конфигурационный файл tlink, 135
косинус угла, 483
кэш
память, 34
трасс, 39

Л

лексемы ассемблера, 89
линейный адрес памяти, 51
логические данные, 194
логические операции, 194
логический сдвиг, 199
логическое исключающее сложение, 195
логическое сложение, 195
логическое умножение, 195
локальная дескрипторная таблица, 56
локальные блоковые метки, 234

М

макроассемблер, 294
макробиблиотека, 298
макрогенерация, 300
макродирективы, 305
макрокоманды, 86, 133, 297
макроопределения, 297
макрорасширение, 300
макрос MAKEINTRESOURCE, 409
мантисса, 458, 479
массив, 269
масштабирование
индекса массива, 272
регистра, 96
масштабный множитель, 70
математический сопроцессор, 21, 448
машинные команды
CISC, 23
RISC, 23
машинный формат команд
сопроцессора, 467
машинный язык, 19
метка, 87, 210
механизм
вызова процедур, 222
макроподстановок, 154
переименования регистров, 36
препроцессорной обработки, 294
микроархитектура, 29
NetBurst, 30, 38
P6, 30, 34
микрооперация, 36
микропрограмма, 19, 60
модификатор
в команде перехода, 215
вызова процедуры, 222
языка, 110
модуль, 325, 328
модульное программирование, 325

Н

непосредственный операнд, 71, 91
нетерминальный символ, 88
неупакованный двоично-десятичный
тип, 113
нечисла, 465
нормализованный вид, 459
нормальная форма Бэкуса–Наура, 86
нуль, 464

О

обработчик
исключений сопроцессора, 504
событий, 443
обратная польская запись
(ПОЛИЗ), 449, 467
объединение, 284
объектный модуль, 129
окна диалога, 420
оконная функция, 369, 398
оконное Windows-приложение, 367
округление результатов работы
команды, 481
операнд, 60, 87, 90
оперативная память, 18, 49
оператор
!, 301
$, 212
%, 301
&, 302
. (точка), 282
: (двоеточие), 211
break, 240
continue, 240
do-while, 241

оператор *(продолжение)*
DUP, 270
for, 241
goto, 240
if-else, 237
inline, 345
MASK, 288
PTR, 148
switch, 237
TYPE, 283
WIDTH, 288
операторы, 97
арифметические, 98
арифметические сравнения, 99
выбора, 237
выделения, 295
замены, 302
именования типа структуры, 102
индексные, 100
итерационного цикла, 241
логические, 99
переопределения
сегмента, 100
типа, 100
получения
сегментной информации, 102
смещения выражения, 102
сдвига, 98
условные, 237
цикла с постусловием, 241
операция распространения знака, 181
описание шаблона структуры, 281
определение структуры, 281
основание системы счисления, 458
особый случай, 454
отладчик TD, 137
отрицание, 194

П

передача аргументов в процедуру, 330
в языках высокого уровня, 342
по адресу, 337
по значению, 336, 355
по ссылке, 355
переменная, 330
перемещаемый операнд, 92
переполнение
мантиссы, 175
разрядной сетки, 172
перерисовка изображений, 413
пересылка битов, 207
переход
косвенный
внутрисегментный, 216
межсегментный, 218
регистровый межсегментный, 218
переход *(продолжение)*
прямой
внутрисегментный, 216
короткий внутрисегментный, 216
межсегментный, 217
поиск элемента в массиве, 276
порт ввода-вывода, 93, 149
порядок числа, 458, 459, 479
постбайт, 64
постфиксная запись, 468
поток, 388
предопределенные имена
@@, 235
@B, 235
@F, 235
предсказание перехода, 23, 32
префикс, 87
блокировки шины, 61
замены сегмента, 61, 102
машинной команды, 61
повторения, 61
программного сегмента, 214
размера
адреса, 61
операнда, 62
смены алфавита, 63
принцип
микропрограммирования, 28
микропрограммного управления, 19
программная модель
процессора, 30, 40
сопроцессора, 448
программные отладчики, 136
пролог, 336, 350
протокол MESI, 35
процедура, 219, 327
процесс, 388
процессор, 18
процессоры Intel, 21
i4004, 21
i8008, 21
i80286, 22
i80386, 22
i80486, 23
i8080, 21
i8086, 21
i8088, 21
Pentium II, 24
Pentium IV, 24
Pentium Pro, 24
Pentium-60, 23
псевдооператор
= (равно), 295
EQU, 295

Р

расширенный программируемый контроллер прерываний APIC
  IC, 23
расширенный формат вещественного числа, 456
регистр
  CWR, 449, 454
  DPR, 450
  EFLAGS/FLAGS, 46, 292
  IPR, 450
  SWR, 449, 453, 500
  TWR, 450, 455
регистровый операнд, 93
регистры
  аккумуляторы, 94
  общего назначения, 44
  процессора, 18
  сегментные, 45, 69
  системные, 43
  слова тегов, 450
  состояния и управления, 46
  состояния сопроцессора, 449
  стека сопроцессора, 455
  тегов, 455
  указателей
    данных, 450
    команд, 450
  флагов, 46
редактор ресурсов, 406
режим работы процессора, 42
  виртуального процессора 8086, 22, 42
  защищенный, 22, 42
  реальный, 42
  системного управления, 42
режимы округления, 481
ресурс Windows-приложений, 406
ресурсный файл, 135

С

сегмент, 50, 86
сегментированная модель памяти, 50
селектор, 56
сигнальное нечисло, 465
синтаксис формального аргумента макроопределения, 302
синтаксическая диаграмма, 86
синус угла, 483
синхронные сообщения Windows, 393, 399
система
  машинных команд, 18, 58
  представления информации
    в компьютере, 194
  счисления, 74
система *(продолжение)*
  непозиционная, 74
  позиционная, 75
системная шина, 18
скалярная архитектура, 31
сканер, 245
слово, 111
служебное слово VARARG, 342, 343
сообщение, 369, 394
сортировка массива, 278
спекулятивное исполнение, 33
специальные численные значения, 463
сплошная модель памяти, 51
спокойное нечисло, 465
среда сопроцессора, 497
стандарт
  IEEE-754, 21, 448
  IEEE-854, 448
стандартные директивы сегментации, 106
стартовый код, 378, 387
стек, 160, 334
стек сопроцессора, 449
страничная модель памяти, 50
структура, 280
  WNDCLASS, 385
  WNDCLASSEX, 385
структурное программирование, 325
суперскалярная архитектура, 23, 31
счетчик
  адреса, 92, 133, 212
  размещения, 93

Т

таблица
  ASCII, 123
  истинности, 194
  перекрестных ссылок, 133
  псевдонимов регистров, 36
таймер, 152.
терминальный символ, 88
тестирование программы, 135
технология HyperThreading, 39
тип
  данных, 110
  операнда, 60
точка входа в программу, 138
трансляция программы, 129
трансцендентные функции, 483
трехходовая суперскалярная конвейерная архитектура, 31, 32

У

указатель на память, 112
упакованные десятичные числа, 458
упакованный двоично-десятичный тип, 113

упрощенные директивы сегментации, 106
условный оператор, 237
устройство
ввода-вывода, 18
выборки/декодирования, 34, 35
диспетчеризации/исполнения, 34, 37
микропрограммного управления, 29
связи с памятью, 35, 38
удаления и восстановления, 37
шинного интерфейса, 22, 34
утилита
cxe2bin, 214
make.exe, 144
nmake.exe, 144
учетверенное слово, 111

Ф

файл
COM, 213
Def, 383
библиотеки, 134
карты, 134
листинга, 131
определений, 135, 402
фактические аргументы, 300, 330
физический адрес, 49
флаг направления df, 252
фон Нейман, 28
форма Бэкуса–Наура, 246
формальная логика, 193
формальные аргументы, 300, 330
функциональная декомпозиция задачи, 325
функция RegisterClass, 385

Х

характеристика, 459

Ц

целое число, 90
целый тип
без знака, 112
со знаком, 111
центральный процессор, 18
цепочечные команды, 249
цепочка, 89, 112
цикл, 230, 240
вложенный, 233
обработки сообщений, 369, 396
с предусловием while, 240
циклический сдвиг, 201

Ч

частичный арктангенс угла, 483
частичный остаток от деления, 490
частичный тангенс угла, 483

Ш

шаблон записи, 286
шина
ISA, 18
PCI, 18
расширения, 18
системная, 18

Э

экранный буфер, 436
эпилог, 336, 352
эффективный адрес, 50, 61

Я

язык ассемблера, 20

Учебное издание

Юров Виктор Иванович

Assembler. Учебник для вузов
2 издание

Главный редактор *Е. Строганова*
Заведующий редакцией *И. Корнеев*
Руководитель проекта *Ю. Суркис*
Литературный редактор *А. Жданов*
Художник *Я. Биржаков*
Иллюстрации *Л. Харитонов, В. Шендерова*
Корректоры *Я. Рощина, Н. Лукина*
Верстка *Л. Харитонов*

Лицензия ИД № 05784 от 07.09.01

Подписано к печати 22.07.03. Формат 70x100 1/16
Усл. п. л. 51,6. Доп. тираж 4 500. Заказ 256
ООО «Питер Принт»
196105, Санкт-Петербург, ул. Благодатная, 67

Налоговая льгота — общероссийский классификатор продукции
ОК 005-93, том 2; 95 3005 — литература учебная.

Отпечатано с готовых диапозитивов
в ФГУП ордена Трудового Красного Знамени «Техническая книга»
Министерства Российской Федерации по делам печати,
телерадиовещания и средств массовых коммуникаций
198005, Санкт-Петербург, Измайловский пр., 29.

КЛУБ ПРОФЕССИОНАЛ



В 1 997 году по инициативе генерального директора **Издательского дома «Питер»** Валерия Степанова и при поддержке деловых кругов города в Санкт-Петербурге был основан **«Книжный клуб Профессионал».** Он собрал под флагом клуба профессионалов своего дела, которых объединяет постоянная тяга к знаниям и любовь к книгам. Членами клуба являются лучшие студенты и известные практики из разных сфер деятельности, которые хотят стать или уже стали профессионалами в той или иной области.

Как и все развивающиеся проекты, с течением времени книжный клуб вырос в **«Клуб Профессионал».** Идею клуба сегодня формируют три основные «клубные» функции:

- неформальное общение и совместный досуг интересных людей;
- участие в подготовке специалистов высокого класса (семинары, пакеты книг по специальной литературе);
- формирование и высказывание мнений современного профессионала (при встречах и на страницах журнала).

КАК ВСТУПИТЬ В КЛУБ?

Для вступления в **«Клуб Профессионал»** вам необходимо:

- ознакомиться с правилами вступления в **«Клуб Профессионал»** на страницах журнала или на сайте **www.piter.com**;
- выразить свое желание вступить в **«Клуб Профессионал»** по электронной почте **postbook@piter.com или по тел. (812) 1 03-73-74**;
- заказать книги на сумму не менее 500 рублей в течение любого времени или приобрести комплект **«Библиотека профессионала»**.

«БИБЛИОТЕКА ПРОФЕССИОНАЛА»

Мы предлагаем вам получить все необходимые знания, подписавшись на **«Библиотеку профессионала».** Она для тех, кто экономит не только время, но и деньги. Покупая комплект - книжную полку **«Библиотека профессионала»**, вы получаете:

- **скидку 1 5%** от розничной цены издания, без учета почтовых расходов;
- при покупке двух или более комплектов – дополнительную **скидку 3%;**
- **членство в «Клубе Профессионал»;**
- подарок - журнал **«Клуб Профессионал».**

Закажите бесплатный журнал «Клуб Профессионал».

